Chapter 3
Processes (part 1)

CS 3423 Operating Systems
National Tsing Hua University

Outline

- Process concept
- Process scheduling
- Operations on processes
- Interprocess communication
- Example IPC
- Client-Server Systems

Objectives

- Introduce the notion of a **process**
  - a program in execution, basis of all computation
- Describe the various features of processes
  - scheduling, creation and termination, communication
- Explore interprocess communication
  - shared memory and message passing
- Describe communication in client-server systems

Process Concept

Program vs. Process

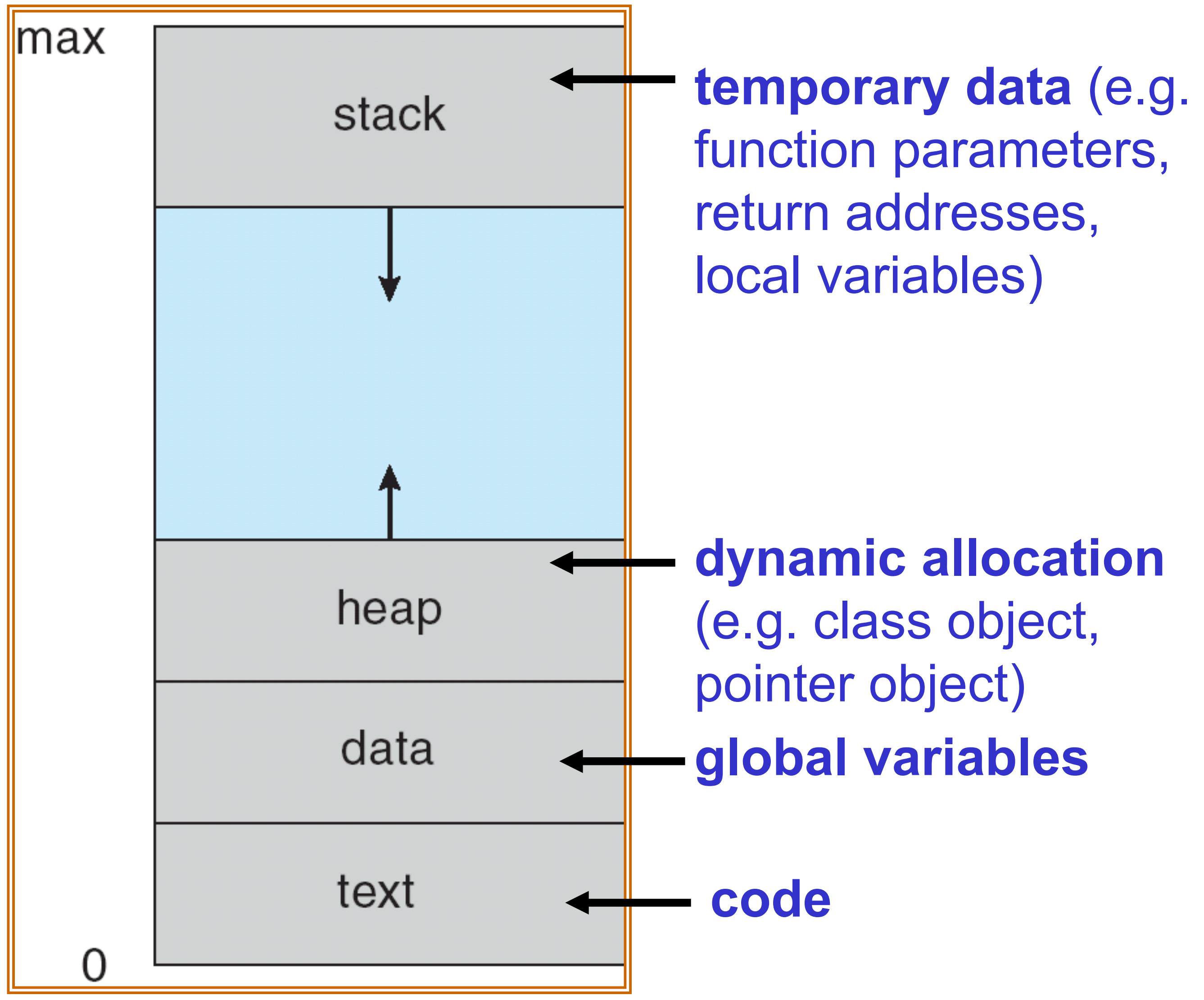
- Program
  - executable code
- Process
  - an instance of a program in execution
  - i.e., has started running; not yet finished
  - possibly multiple instances of a program (e.g. multiple users running same email client on the same computer)

Terminology

- "process"
  - standard usage nowadays = instance of a running program
- "job"
  - synonym with "process" , but "process" is preferred
  - from scheduling literature (Operations Research)
    "job-shop scheduling"
- "task"
  - informal word for process ("multitasking"), possibly from user's point of view of "a unit of work that needs to be done"
  - from real-time systems, maybe lighter weight than process

Process in Memory

- **code** segment (“text section”)
- **data** section, for global vars
- **stack**: for (auto) local vars of functions, parameters passed to function call, return address
- **heap**: dynamically allocated variables (incl. objects)
- program state: (**program counter**, **registers**)
- a set of associated resources (e.g., open file handles)

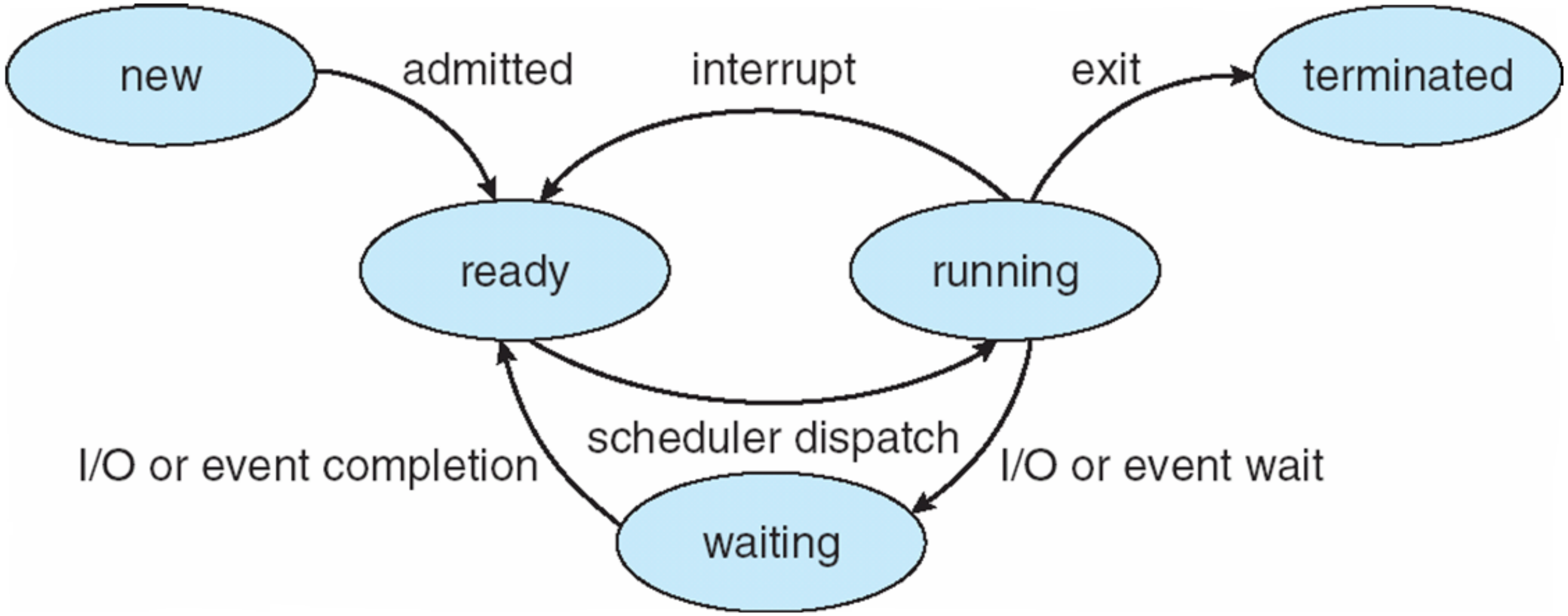

max
stack
heap
data
text
0
temporary data (e.g. function parameters, return addresses, local variables)
dynamic allocation (e.g. class object, pointer object)
global variables
code

Process state

- New
  - the process is being created (by the OS)
- Ready
  - the process is in memory, can be assigned to a processor, but is not currently running.
- Running
  - the process's instructions are being executed by the processor
- Waiting
  - the process is waiting for some event ("blocked"), could be I/O
- Terminated
  - the process has finished execution; its space can be reclaimed

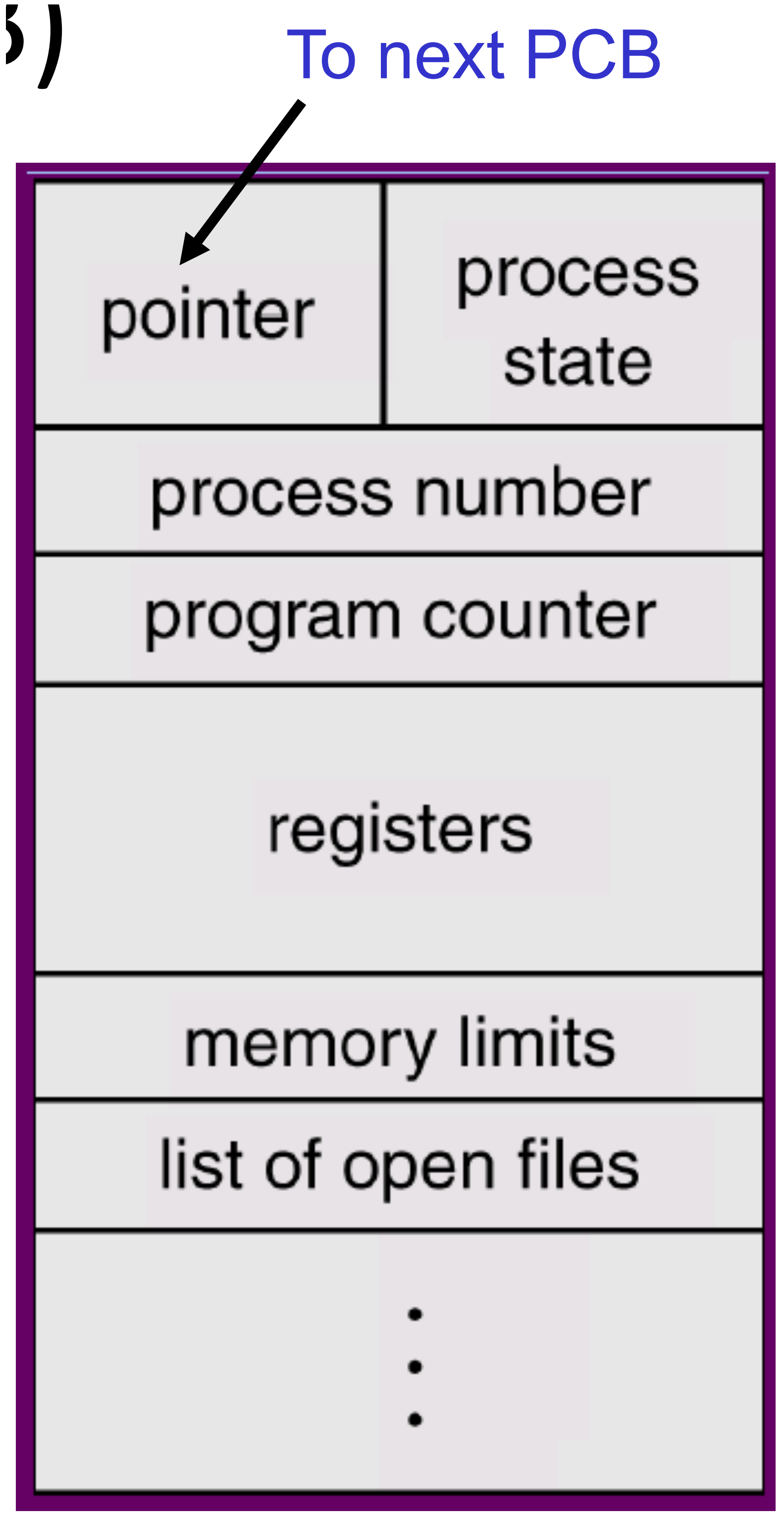
Diagram of Process State

- Only one process is Running on any processor at any time
- However, several processes may be Ready or Waiting



Process Control Block (PCB)

- Information associated with each process
  - also called **task control block**
- Process state – RUNNING, WAITING, etc
- Program counter, CPU registers
- CPU scheduling information
  - priorities, scheduling queue pointers
- Memory-management information
  - memory allocated to the process
- Accounting information –
  - CPU used, clock time elapsed since start, time limits
- I/O status information –
  - I/O devices allocated to process, list of open files



Threads

- aka “lightweight processes”
  - a basic unit of program execution
  - Multiple threads may belong to one process
- Threads of a given process share…
  - code section, data section, OS resources
- Each thread has its own…
  - thread ID, program counter, register set, and stack

Review (1)

- Definition of a process?
- Difference between process and thread?
- What are possible Process States?
- What is a PCB, and what is its content?
- How does Context Switch work?

Process Scheduling

Process Scheduling

- OS Purpose
  - Multiprogramming: maximize CPU utilization (i.e., runs some process at all times)
  - Time-sharing: interactivity, short latency (i.e., switches CPU frequently so user can interact with programs)
- Scheduling
  - OS decides when to run each process and for how long

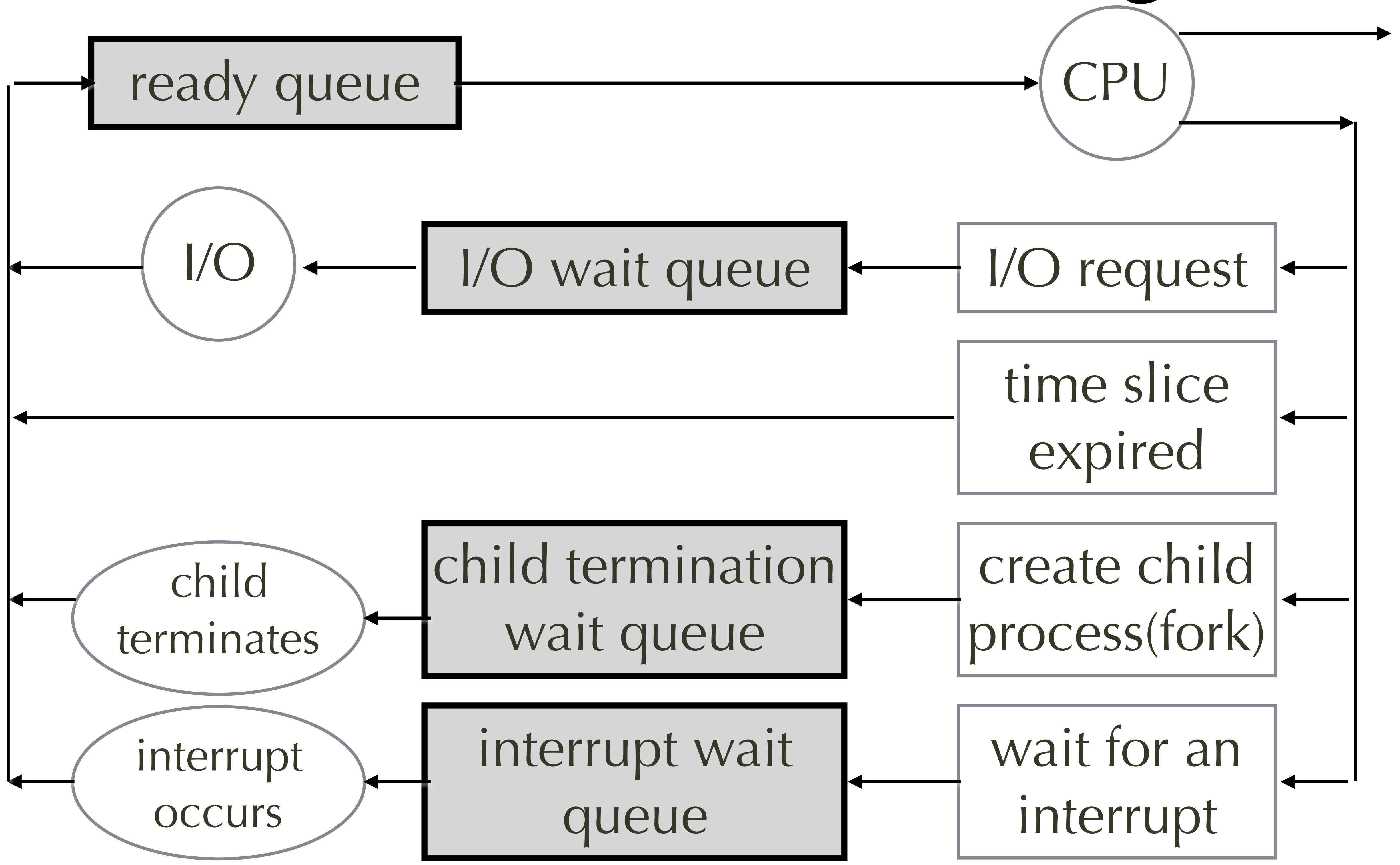
Scheduling terms

- Degree of multiprogramming
  - number of processes kept in memory
    (as opposed to **swapped out** of main memory to disk)
- I/O-bound processes
  - spends more time doing I/O than computing
  - many short CPU **bursts**
- CPU-bound processes
  - spends more time doing computation
  - few but long CPU bursts

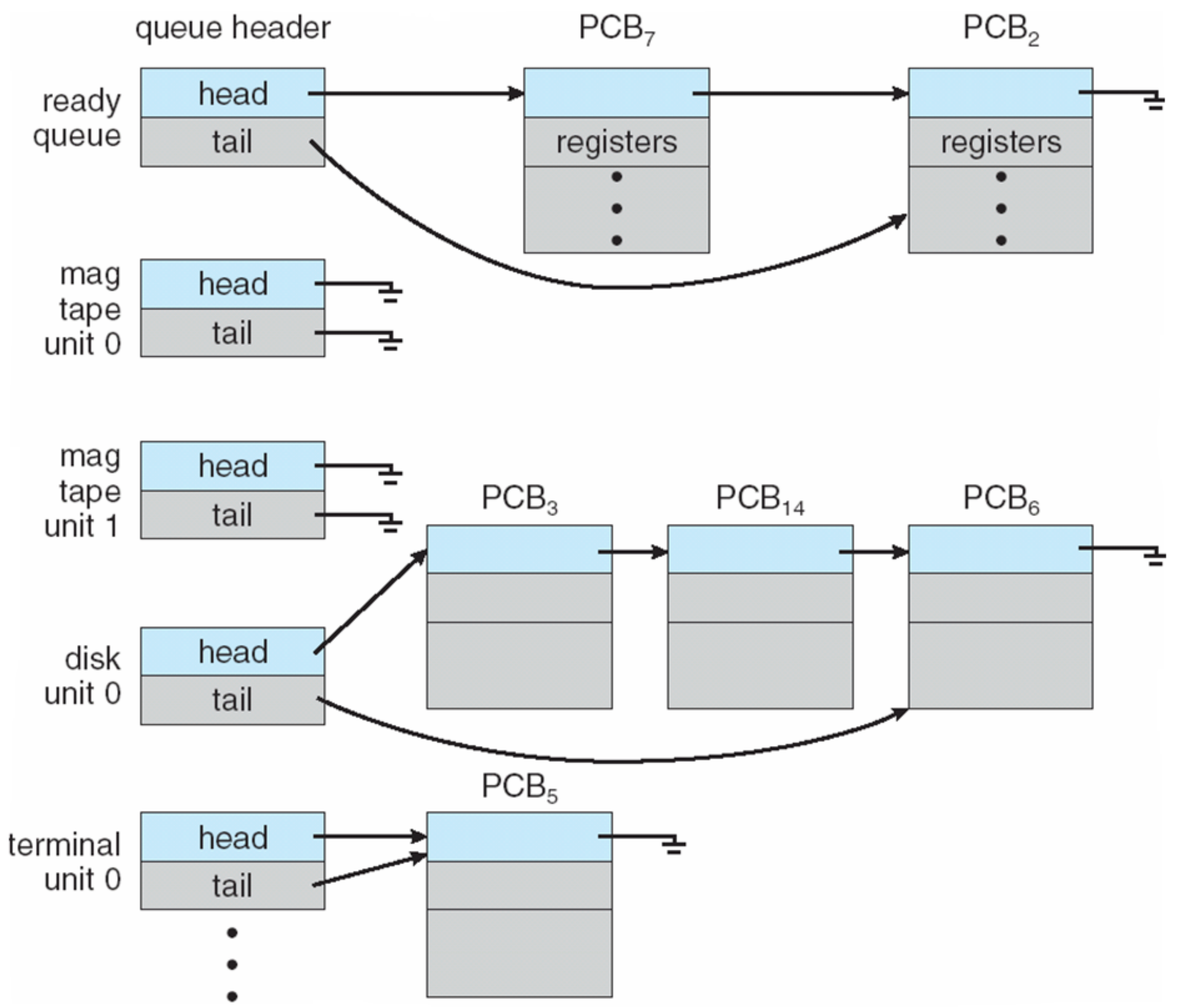
Process Scheduling Queues

- Processes can migrate between different queues (i.e., switch among states)
- Job queue (NEW state)
  - set of all processes in the system
- Ready queue (processes in READY state)
  - set of all processes residing in main memory, ready and waiting to execute
- I/O queue
  - set of process (in WAIT state) waiting for an I/O device

"Queue Diagram" representation of Process Scheduling

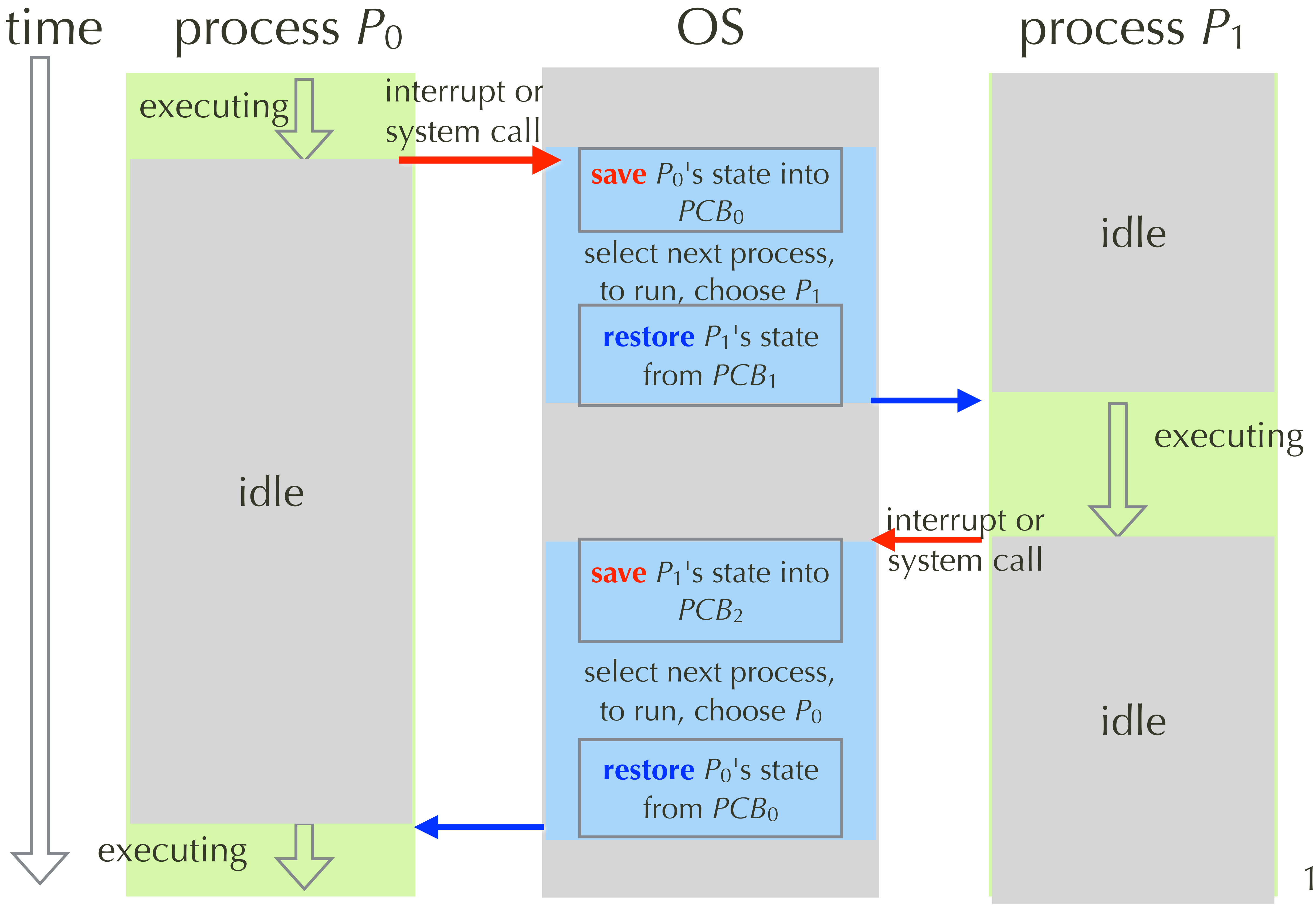


Process Scheduling Queues



I/O queue

Context Switch


time
process P_0
OS
process P_1
executing
interrupt or system call
save P_0's state into PCB_0
select next process, to run, choose P_1
restore P_1's state from PCB_1
idle
idle
executing
interrupt or system call
save P_1's state into PCB_2
select next process, to run, choose P_0
restore P_0's state from PCB_0
idle
executing

Context Switch

- Switch to a different process to run
    - Kernel saves the state of currently running process
    - Kernel restores the saved state of the target process
- Overhead
    - time spent by OS, not productive time for the user
    - switching time: 1-1000 ms, depending on memory speed, #registers

Hardware support for context switching

- instruction for store/load multiple registers
  - ARM instructions load, store, push, pop multiple regs

```
LDM   {r2, lr}    ;; (load multiple)
STM   {r2, lr}    ;; (store multiple)
PUSH  {r0,r4-r7}
POP   {r0,r10,pc}
```

  -- all work on multiple regs

- Register windows
  - Sun SPARC ISA uses sliding register windows
  - 8051 has four register banks

Multitasking in Mobile Systems

- UI provides important hint on what needs to be scheduled
  - Single **foreground** process -- controlled via user interface
  - Multiple **background** processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Purposes
  - Saves power, improve responsiveness
- Android runs foreground and background, with fewer limits
  - A background process uses a service to perform tasks
  - A service can keep running even if the background process is suspended
  - A service has no user interface; is small in memory use

Review: Context Switch

- CPU switches to another process
  - OS must save the state (register, etc) of the old process
  - OS loads the saved state for the new process via a context switch
  - PCB: representation of Context of a process
- Overhead reduction
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
  - efficient coding and data structure

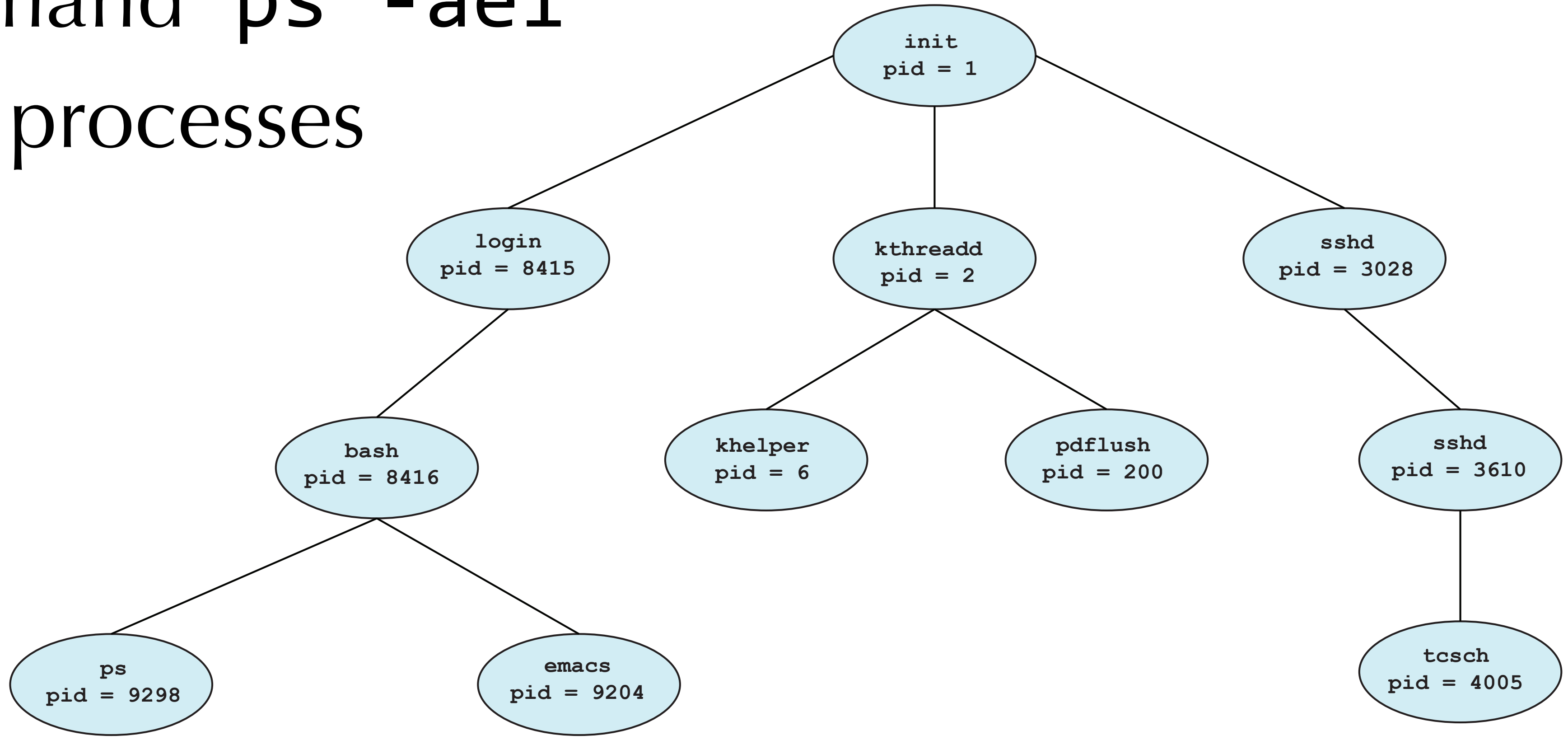
Operations on Processes

Operations on processes

- process **Creation**
  - fork() = clone, exec() = replace
- process **Termination**
  - exit() = voluntary, abort() = involuntary
  - wait() = sync with terminating child process
- in addition to process switching
  - save / restore state, pick next to run (scheduling)

Process Creation

- parent process creates children processes
  - family tree
- each process has a unique identifier (pid)
  - Unix command `ps -ael` lists active processes



Options of Process Creation

- Sharing options:
  - share **all** resources
  - child shares **subset**
  - **no sharing**
- Execution options
  - concurrent execution
  - parent waits until all children terminate

Address Space Options

- child is a duplicate of parent
  - child runs the same program image as parent
  - communicate via shared variable
- child program is not a duplicate
  - program replaced by a newly loaded program
  - communicate via message passing

`fork()` system call

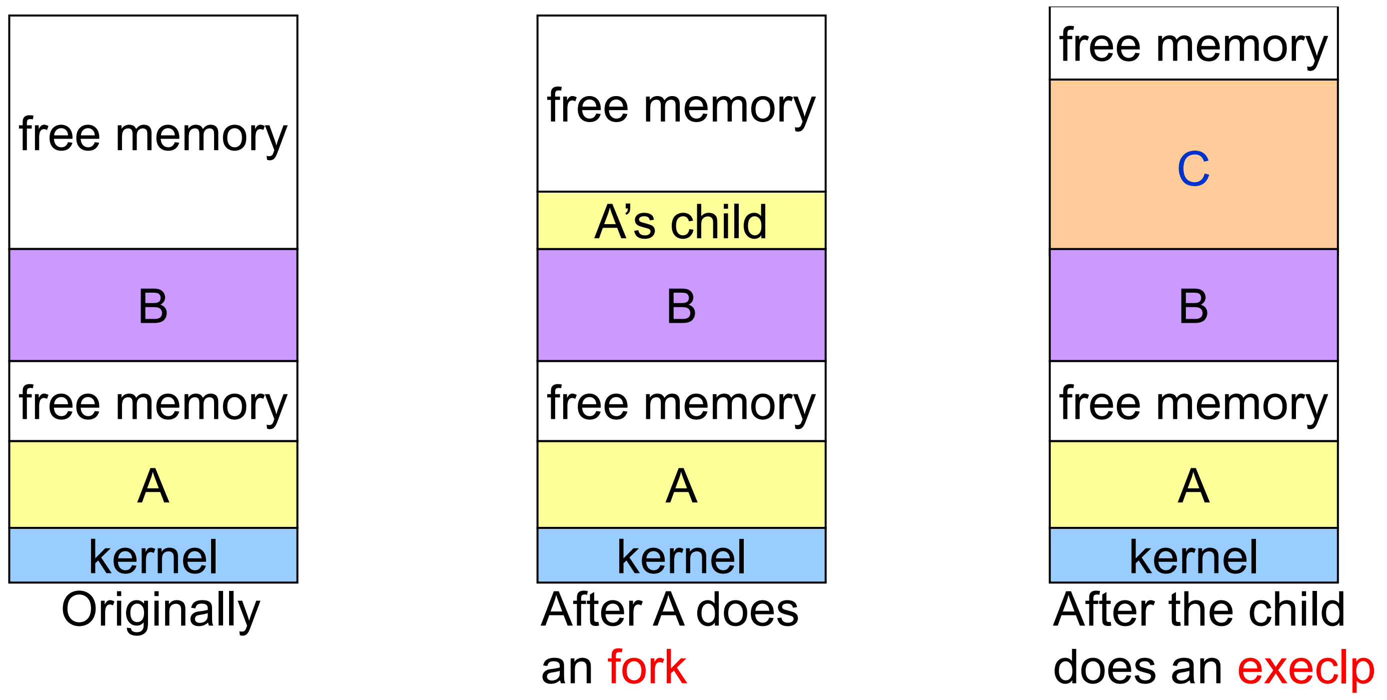
- parent clones itself
  - child process duplicates address space of the parent (i.e., a copy)
  - child and parent execute concurrently after `fork`
- return value of `fork()`
  - child gets `0`
  - parents gets `pid` of child

`exec()` system call

- `exec()`:
  - **replaces** process itself with specified program (in args)
  - **restart** process
- Return value?
  - If successful, `exec()` does not return! because it runs the new program
  - But if error (e.g., program not found) then returns `-1` with error code in a global variable `errno`
- API variants of `exec`:
  - `execlp()`, `execl()`, `execle()`: *path*, *arg*0, *arg*1, ..., `NULL`
  - `execv()`, `execvp()`: *path*, *argv*`[]`
  - `execvP()`: *file*, *searchpath*, *argv*`[]`

Process Creation in Unix/Linux Data memory

- Old implementation:
  - child is a full copy of parent
- Current implementation: copy-on-write
  - no need to store extra copy of same data;
  - saves work of copying => both more efficient

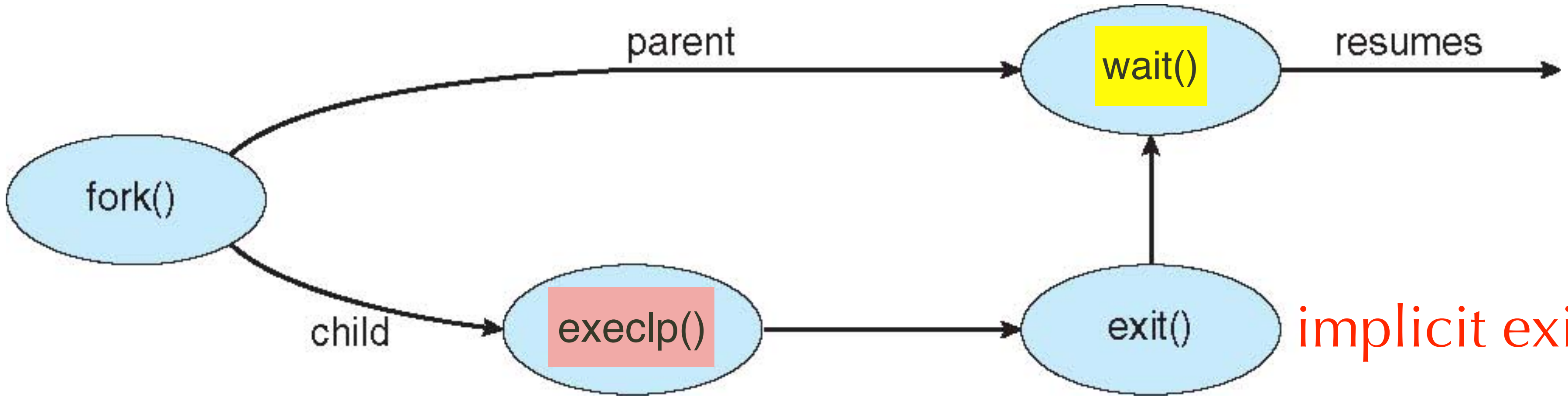


Unix Example

```c
#include <stdio.h>
#include <unistd.h> // for fork()
#include <sys/wait.h> // for wait()
void main() {
  pid_t A = fork();
  if (!A) { // child
    printf("child\n");
    execlp("/bin/ls", "ls", NULL);
  } else { // parent
    printf("parent\n");
    int   status;
    pid_t pid=wait(&status);
    printf("child %d done\n", pid);
  }
  printf("process %d ends\n", A);
}
```

Output:

```
parent
child
a.out hello.c readme.txt
child 32185 done
process 32185 ends
```



implicit exit by child

Shell example

- Parses command line
    - extract program name and arguments
- calls `fork()`
    - to create new process for new program
- Child process calls `exec()`
    - to load in new program, becomes new program
- Parent:
    - can either continue running shell or `wait()` for child to finish

Process Termination

- option 1: voluntary
  - `exit(`*`status`*`)`: for child to finish & return exit status to parent
  - could be implicit exit upon return from `main()`
- option 2: involuntary (killed)
  - `kill(`*`pid, sig`*`)`: parent terminates child process by pid
  - Why? (1) child exceeds resource quota, (2) task no longer needed, (3) OS may have **cascaded termination** policy
- OS clean-up:
  - OS reclaims all resources: memory, open files, I/O buffers
  - cascaded termination: parent dies => kill all its children (recursive)

Process Termination

- `wait()` system call
  - called by parent to wait for one of its child processes to terminate
  - get that child's return status (exit code) `pid = wait(&status);`
  - OS won't release (recycle) child pid and table entry till parent calls `wait()`!
- zombie process
  - dead child process that died before its parent calls `wait()` to find out...
  - zombie pid released when parent calls `wait()`
- Orphan process:
  - a child process (alive) whose parent died
  - Solution: an ancestor process could call `wait()` to collect orphans
    Root process: `init` (traditional Unix) or `systemd` (Linux)

Interprocess Communication (IPC)

Multiple processes

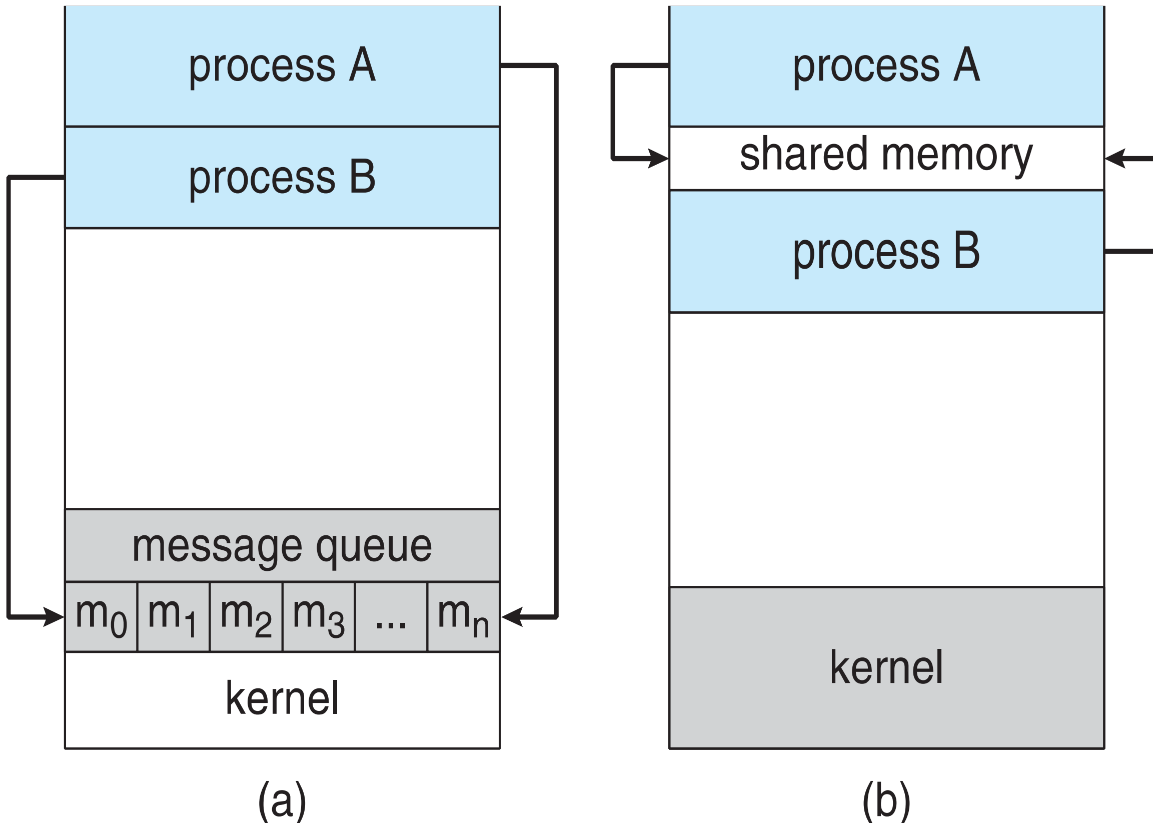
- Communicate or run independently?
  - independent: no resource sharing other than running on same processor
  - communicating processes or threads: exchange data
- reasons for IPC
  - sharing data
  - speedup (multiple processors only)
  - convenience, modularity

Communication methods

- Shared memory
  - requires more careful user **synchronization**
  - implemented by memory access, (i.e., read/write) faster speed
  - doesn't work across machines
- Message passing
  - **send**(msg), **receive**(msg) as system calls
  - no conflict; call may **block**; more efficient for smaller data
  - on same machine or different machines

message passing

shared memory



Interprocess Communication

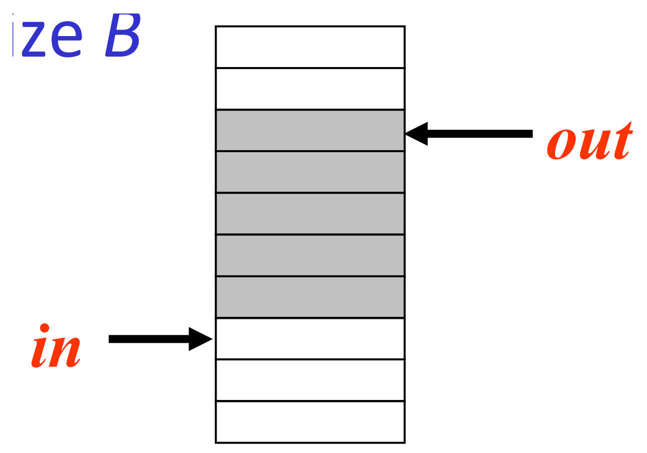
- IPC Models
  - Shared Memory
  - Message passing
- Examples: Shared memory
  - POSIX
- Examples: Message Passing
  - Mach IPC, Pipes
  - Sockets vs. Remote procedure calls

Shared Memory

- Establishing a region of shared memory
  - same address space or different spaces but mapped by OS
  - Doesn't work across machines!
- Used for faster performance
  - no need for data copying; just work on shared data
  - OS involved only during setup, but not during actual read/write!
- Need to determine the form of data and location
  - text or binary, struct, semantics
- Ensure data not written simultaneously inconsistently
  - synchronize by locking or scheduling

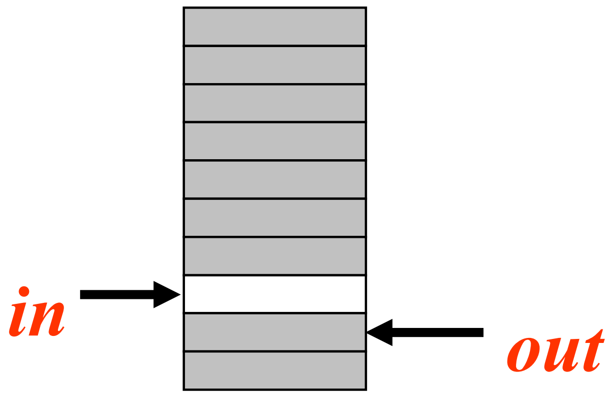
Problem of Producer-Consumer

- Producer-Consumer loop
  - Producer outputs data, Consumer inputs data
- Possible use of buffer: queue (FIFO) with size B
  - in-pointer: next free position
  - out-pointer: position of first available
  - FIFO empty when in == out
  - FIFO full when (in+1)%B == out
  - This allows at most B-1 items in the queue, since one can't tell if the buffer is empty or full.
- Constraints: bounded vs unbounded buffer



Pseudocode for Shared memory Producer

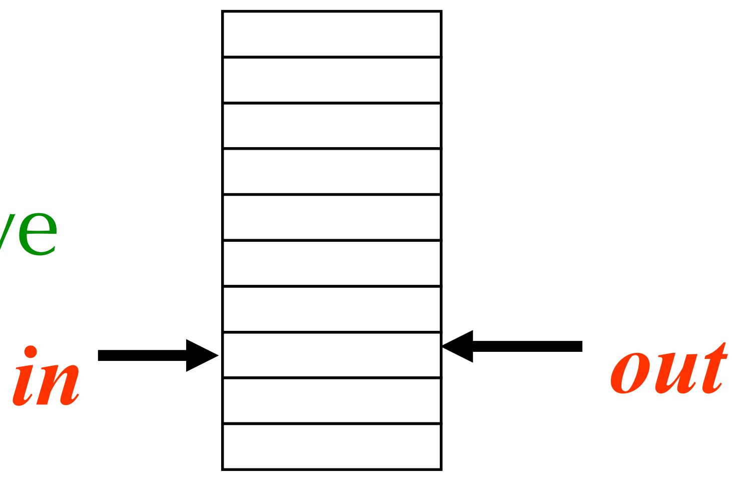
```
• item next_produced; // item is a data type
  while (true) {
    next_produced = make_item();
    while(((in+1)%BUFFER_SIZE)==out) {
      // buffer is full, so we wait (polling)
      // assume consumer can run when
      // producer is polling.
      yield; // cooperative; nothing if preemptive
    }
    buffer[in] = next_produced;
    in = (in+1)%BUFFER_SIZE;
    // in is modified only by producer
  }
```



(in+1)%BUFFE_SIZE
== out
means full

Pseudocode for Shared memory Consumer

```
• item next_consumed; // item is a data type
  while (true) {
    while (in==out) {
      // buffer is empty, so we wait (polling)
      // assume producer can run when
      // the consumer polls.
      yield; // if cooperative; nothing if preemptive
    }
    next_consumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    // out is modified only by consumer
    use_item(next_consumed);
  }
```



in == out
means empty

Interprocess Communication

- IPC Models
  - Shared Memory
  - Message passing
- Examples: Shared memory
  - POSIX
- Examples: Message Passing
  - Mach IPC, Pipes
  - Sockets vs. Remote procedure calls

Message-Passing Communication

- Mechanism for processes to communicate **and synchronize** their actions
  - processes communicate without resorting to shared variables
- Two fundamental operations for IPC (pseudocode)
  - `send(h, msg)` // could be fixed- or variable message size
  - `receive(h, &buf)` // `#` bytes, status may be additional params
- Assumption before communicate
  - processes need to establish a communication link first!!!
  - *h* (as in `send(h, msg)`, `receive(h, &buf)`) could be a "handle" to the link, a process, or mailbox

Communication Links in Message Passing

- How are links established?
- Can a link be associated with > 2 processes?
- Between two processes, how many links can there be? (multiplicity)
- What is the link capacity?
- Data length: fixed- or variable-sized msg?
- is the link unidirectional or bidirectional?

Implementation of Communication Links

- Physical link
  - shared memory
  - hardware bus
  - network
- Logical
  - Naming: direct or indirect? symmetric or asymmetric naming?
  - Synchrony: blocking or nonblocking? (synchronous vs. asynchronous)
  - Buffering: automatic or explicit buffering?
  - Data Copying: send by copy or by reference?

Direct (message passing) Communication

- Processes must name each other explicitly
  - `send(P, message)`: send *`message`* to process *`P`*
  - `receive(Q, &buf)`: receive a msg from process Q into *`buf`*
- Properties of communication link
  - Links are established automatically (or hardwired)
  - One link is associated with exactly two processes, and between a pair of processes, there exists exactly one link
  - They may be bidirectional (usual) or unidirectional

Process symmetry

- symmetric
  - sender and receiver name each other
  - send(P, msg)   ....   receive(Q, &buf)
- asymmetric:
  - sender names the target process to send to
  - receiver receives from ANY process and gets sender ID

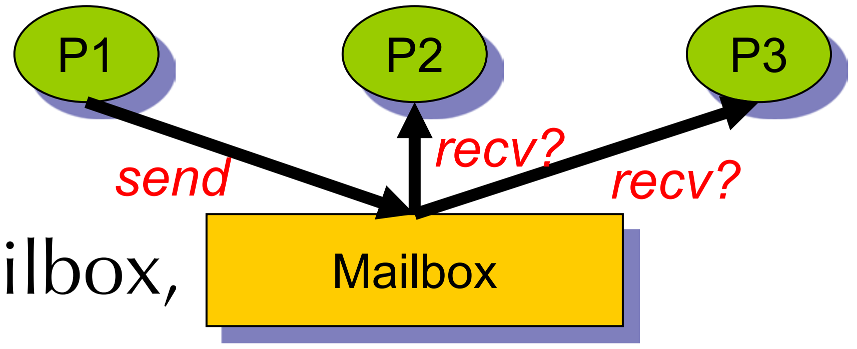
Producer-consumer by Direct Communication

- /* producer */

  -
    ```
    while (1) {
      send(consumer, nextProduced);
    }
    ```
- /* consumer */

  -
    ```
    while(1) {
      receive(producer, nextConsumed);
    }
    ```
- Issue: Limited modularity
  - if name of a process changed, all old names need to be updated

Indirect Communication



- Mailbox, aka ports
  - send message to mailbox or receive from mailbox, instead of direct send-receive
  - Each mailbox has a unique ID
  - processes must share a mailbox in order to communicate
- Link properties
  - Link established only if processes share a common mailbox
  - a link may be associated with multiple processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bidirectional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives
  - send(A, msg) // send msg to mailbox A
  - receive(A, &buf) // receive a msg from mailbox A

Indirect Communication

- What happens when mailbox is shared?
  - P_1, P_2, P_3 share mailbox A
  - P_1 sends; P_2 and P_3 both receive
  - Who gets the message?
- Possible options (OS dependent)
  1. Allow a link to be associated with at most two processes
  2. Allow only one process at a time to execute a receive() operation
  3. Allow the system to select arbitrarily the receiver
  4. Sender is notified who the receiver was.

Synchrony in Messaging

- Blocking (“synchronous”) call:
  - send/receive does not return till done
    => how regular functional calls work
- Nonblocking (“asynchronous”) call:
  - send/receive returns immediately,
    even before the communication is completed!!
  - a separate call to check if done (like polling)
  - may also use a callback for notification!

Synchrony in send/receive

- Blocking send:
  - sender is blocked until the message is received by the receiver or mailbox
- Blocking receive:
  - receiver is blocked until a message has arrived and can be received
- Nonblocking send:
  - sender writes message to a buffer and continues operation without waiting for send to complete => buffer is required!
- Nonblocking receive:
  - sender receives either an arrived (and queued) memory or receives no message, but does not block in either case.

Buffer and Synchrony

- Zero buffer
  - blocking send, blocking receive (*rendezvous*)
    => earlier one blocks until the later one ready to exchange
- Bounded buffer
  - sender is blocked if buffer is full; else not blocked
  - receiver blocked if buffer is empty; else not blocked
- Unbounded buffer
  - sender never blocks; receiver blocks only if buffer empty

Review (3)

- Shared memory vs Message Passing
- Direct vs Indirect message-passing
- Blocking vs Nonblocking