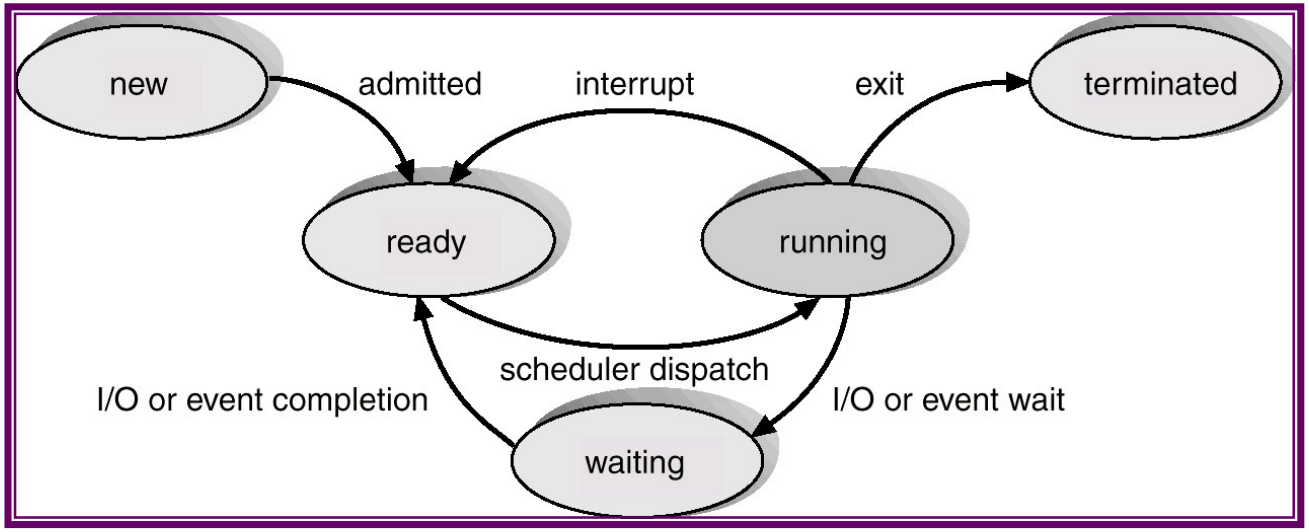
Announcements

- Office hours moved before class hours (15:00-16:00).
- Christine available on Fridays to assist with English.
- Assignment 1 posted; due Friday by 16:00.
- Choose teams for Windows 2000/Linux coverage.
- Last lecture:
  - 100 pages * 50% comprehension = 50 pages digested.
  - Only breadth-oriented lecture; all others depth-oriented.
  - Slides posted (90 slides).
  - Questions on material covered?
- This lecture:
  - 50 pages hence 100% comprehension expected!
  - Slides posted (40 slides).
  - Will cover enough material to do over 50% of assignment 1.
  - Questions on material to be covered?

Chapter 4: Processes

- Process Scheduling.
- Operations on Processes.
- Cooperating Processes.
- Interprocess Communication.
- Communication in Client-Server Systems.

Process States



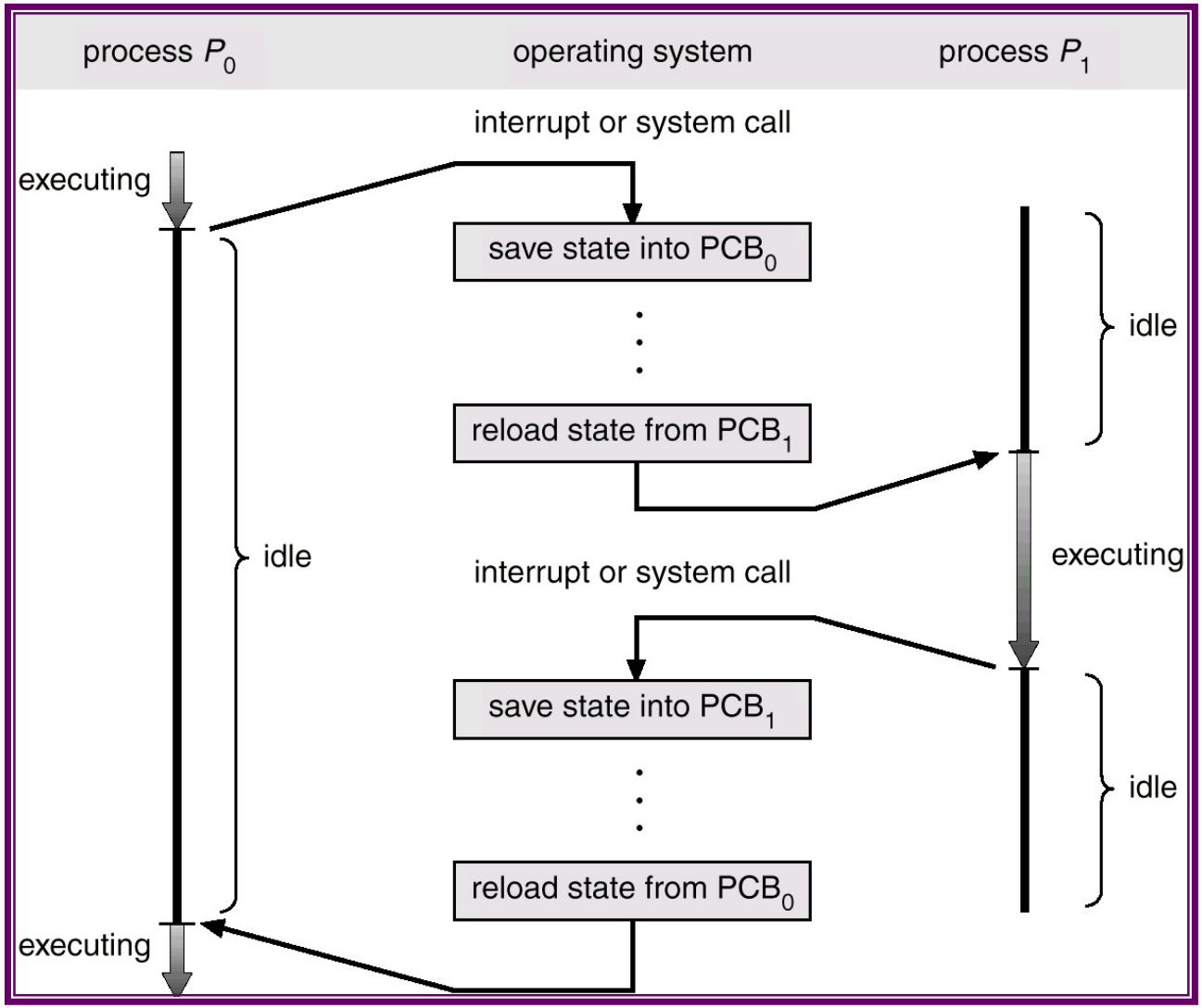
- **new**: The process is being created. Very short-term state.
- **running**: Process on CPU. Instructions being executed.
- **waiting**: The process is waiting for some event to occur.
- **ready**: The process is waiting to be assigned to a CPU.
- **terminated**: The process has finished execution. Doesn't disappear until another process reads its exit status. Listed as <defunct> on UNIX `ps` (a.k.a. *zombie* process).

Process Control Block (PCB)

Information associated with each process:

- Process state.
- Program counter.
- CPU registers.
- CPU scheduling information.
- Memory-management information.
- Accounting information.
- I/O status information.

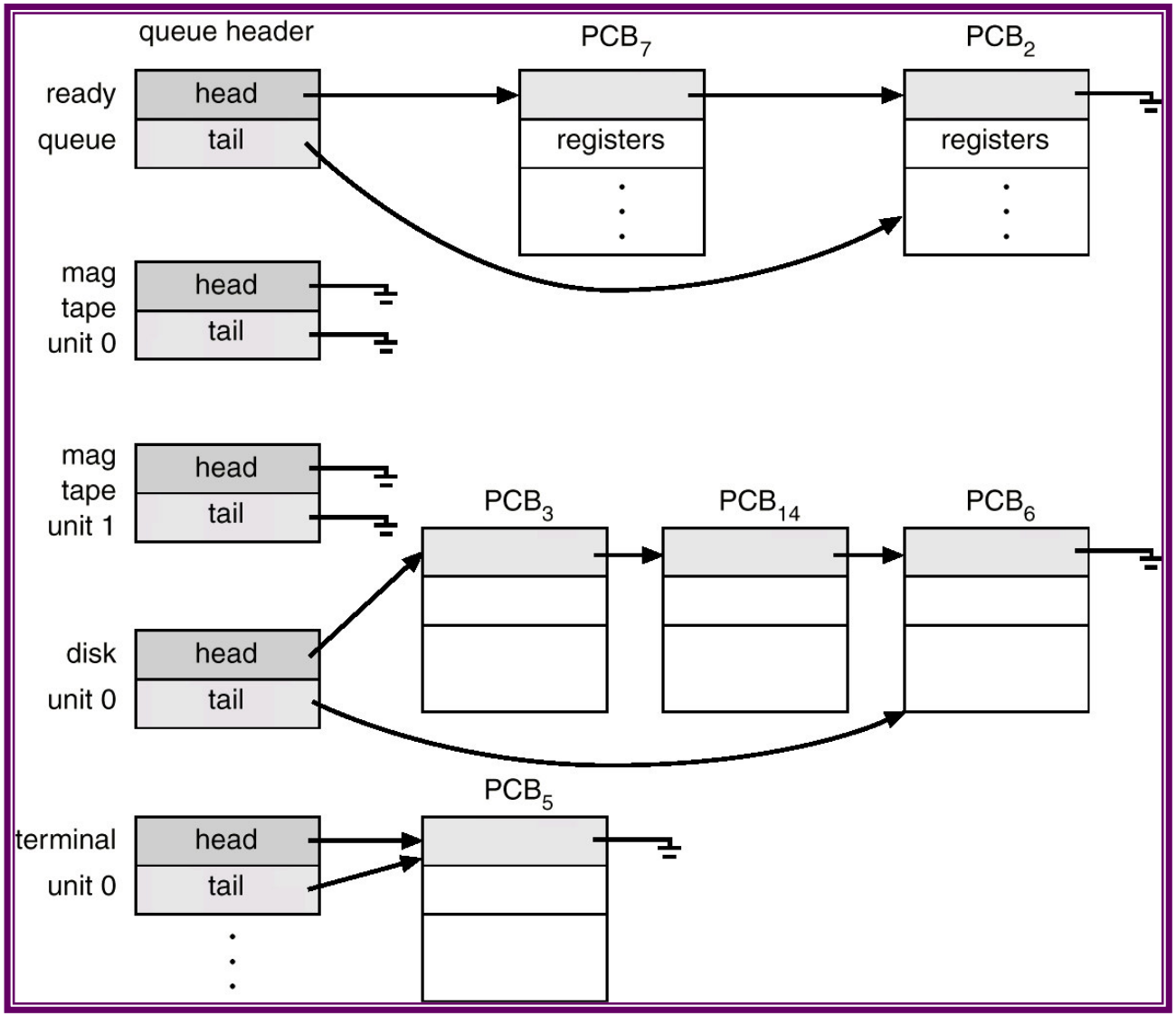
CPU Switch From Process to Process



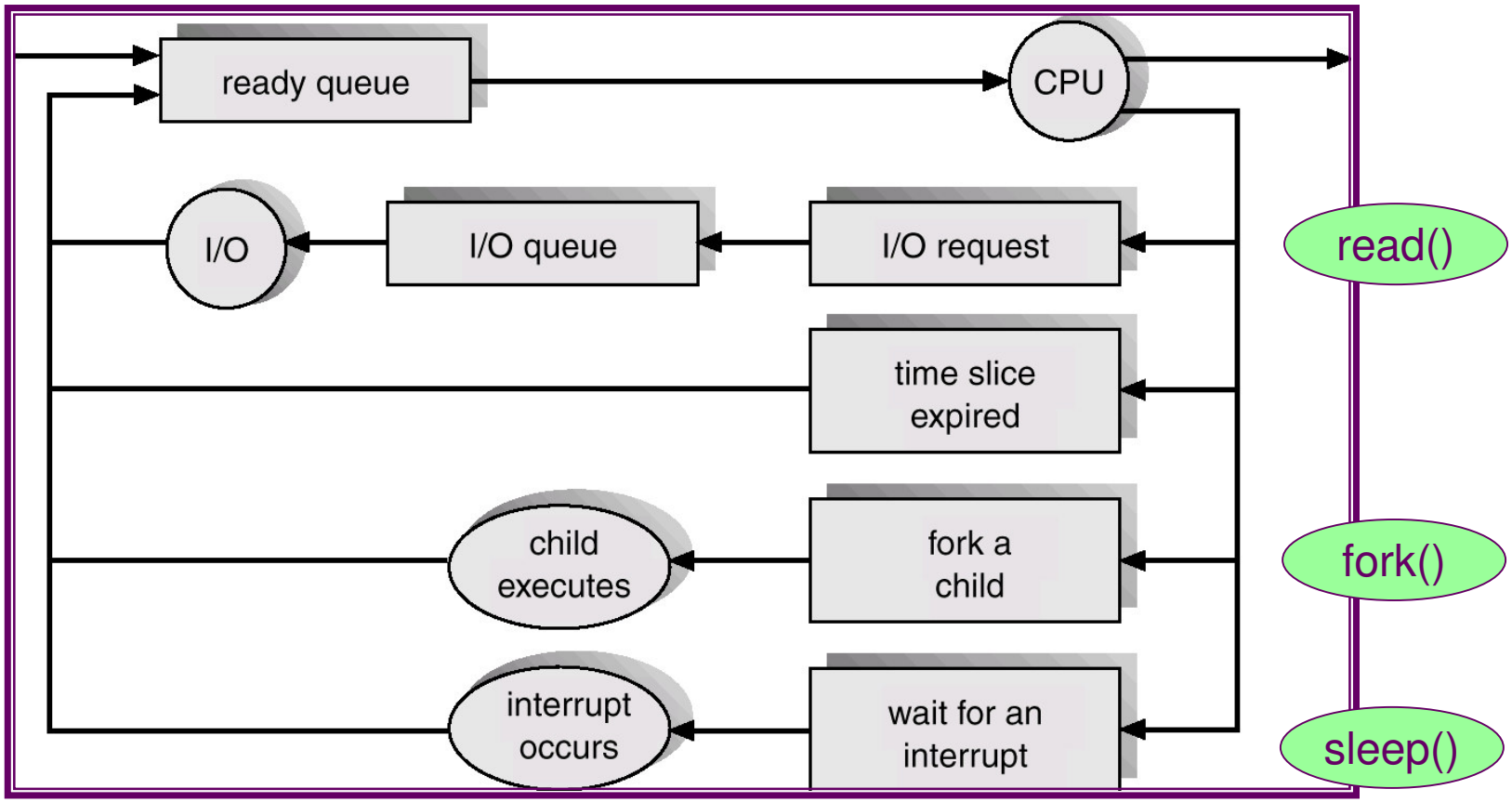
Process Scheduling Queues

- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

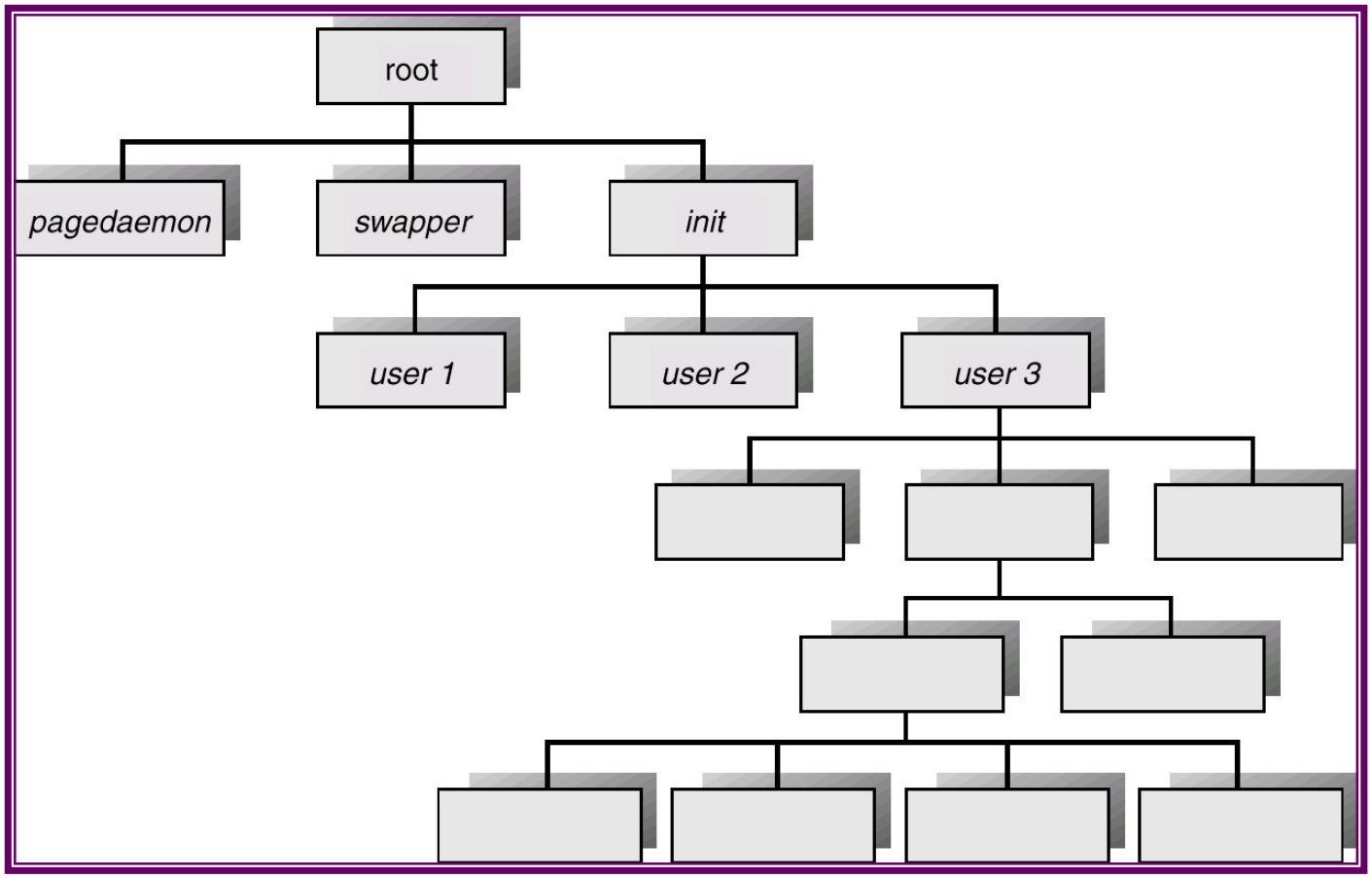
- Long-term (or job scheduler):
  - which process should be started from a program on disk, and brought into the ready queue.
  - invoked infrequently so can be slow.
  - controls degree of multi-programming, i.e. # of processes.
- Medium-term:
  - which process should be moved from memory back to disk, in order to reduce context-switching overhead. “Fixes” bad decisions of long-term scheduler.
  - invoked infrequently so can be slow.
  - controls degree of multi-programming, i.e. # of processes.
- Short-term (or CPU scheduler):
  - which process should be executed next on CPU.
  - invoked very frequently (milliseconds) so must be fast.

Context Switch

- Processes can be described as either:
  - I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts. Need context switch often at beginning & end of I/O.
  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts. Need also context switch often to prevent process from excluding all others from using CPU.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Process Creation

Parent process creates children processes, which, in turn create other processes, forming a tree of processes.



Process Creation (Cont.)

- Resource sharing choices:
  - Share all resources.
  - Share no resources.
  - Children share subset of parent's resource. Specify whether each resource is *inheritable* when creating it.
- Execution:
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- Address space:
  - Child duplicate of parent.
  - Child has a program loaded into it.

Processes in UNIX

- **fork** system call creates new process: child sees result 0, parent sees child process ID.
- **exec** system call often used after a **fork** to replace memory space of child process with a new program.

```
int pid=fork();
if (pid<0) {                          /* error */
        /*print error */ }
else if (pid==0) {                    /* child */
        execlp("/bin/ls","ls",NULL); }
else {                                /* parent */
        int status;
        wait(&status); }              /* wait for child */
```

Process Termination

- Process executes last statement and asks the operating system to terminate itself (**exit**).
  - Output status from child to parent (via **wait**): if child ends with exit(3), parent sees WIFEXITED(status)!=0 and WEXITSTATUS(status)=3.
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**) via signal (kill()).
  - WIFEXITED(status)==0.
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting. Operating system options:
    - Stop child. Cascading termination: grandchildren also abort, etc.
    - Child adopted by grandparent.

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process. Usually siblings.
- *Cooperating* process can affect or be affected by the execution of another process. Usually parent/child.
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - Example: Word prints docs to printer spooler.
  - *Unbounded-buffer:* assumes buffer size unlimited.
  - *Bounded-buffer:* assumes that there is a finite, fixed buffer size.

Bounded-Buffer – Shared-Memory Solution

- Shared data:

```
#define B_SIZE 5
char buffer[B_SIZE];
int pcount=0, ccount=0;
```

- Producer:

```
int in=0;
while (1) {
    /* produce an item in char nextProduced */
    while (pcount-ccount==B_SIZE)
            /* wait while buffer full */;
    buffer[in]=nextProduced;
    pcount++; in=(in+1)%B_SIZE; }
```

- Consumer:

```
int out=0;
while (1) {
    while (pcount==ccount)
            /* wait while buffer empty */;
    nextConsumed=buffer[out];
    ccount++; out=(out+1)%B_SIZE;
    /* consume the item in char nextConsumed */ }
```

Bounded-Buffer – Operation

time	in	pcount	out	ccount	b[0][1][2][3][4]	
0	0	0	0	0	aaaaa	consumer waits
1	1	1	0	0	Uaaaa	
2	2	2	0	0	UUaaa	
3	2	2	1	1	aUaaa	
4	3	3	1	1	aUUaa	
5	3	3	2	2	aaUaa	
6	3	3	3	3	aaaaa	consumer waits

a: slot available
U: slot in use

11	3	8	3	3	UUUUU	producer waits

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions without resorting to shared variables. But same concept as producer/consumer.
- IPC facility operations:
  - **send**(*message*) – message size fixed or variable.
  - **receive**(*message*).
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them.
  - exchange messages via send/receive.
- Two techniques:
  - Direct: P names Q as receiver. Think email.
  - Indirect: P, Q (and R, etc.) share mailbox. Think newsgroups.

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P.
  - **receive**(*Q, message*) – receive a message from process Q.
- Properties of communication link:
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - Link usually bi-directional, or can have two unidirectional links (P-to-Q, Q-to-P).

Indirect Communication

- Messages directed and received from mailboxes (ports).
- Operations:
  - Create/delete mailbox.
  - **send**(*A, message*) – send message to mailbox A.
  - **receive**(*A, message*) – receive message from mailbox A.
- Properties of communication link:
  - Link established only if processes share a common mailbox.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links (mailboxes).
  - Link may be unidirectional or bi-directional: processes decides who reads/writes.

Indirect Communication

- Mailbox sharing ambiguity:
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?
- Solutions:
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
  - Everybody gets a copy.

Synchronization

- Message passing may be either blocking or non-blocking:
  - **Blocking** is considered **synchronous.**
  - **Non-blocking** is considered **asynchronous.**
  - **send** and **receive** may be blocking or non-blocking.
- Queue of messages attached to the link:
  1. Zero capacity – 0 messages.
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite number of messages.
     Sender must wait if link full.
  3. Unbounded capacity – infinite number of messages.
     Sender never waits.

Client-Server Communication

- Sockets.
- Remote Procedure Calls.
- Remote Method Invocation (Java).

Sockets

- A socket is defined as an *endpoint for communication*.
- Concatenation of IP address (machine) and port (application).
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8.**
- Communication consists between a pair of sockets:
  - Server socket hand-picked.
  - Client socket automatically assigned.
- Key element of assignment 1.

Sockets example

- Server on 127.0.0.1 5135:

```
ServerSocker socket=new ServerSocker(5135);
while (true) {
    Socket link=socket.accept();
    int s_in=link.getInputStream().read();
    link.getOutputStream().write(s_in+1); }
```

- Client:

```
Socket link=new Socket("127.0.0.1",5135);
link.getOutputStream().write(c_out);
int c_in=link.getInputStream().read();
```

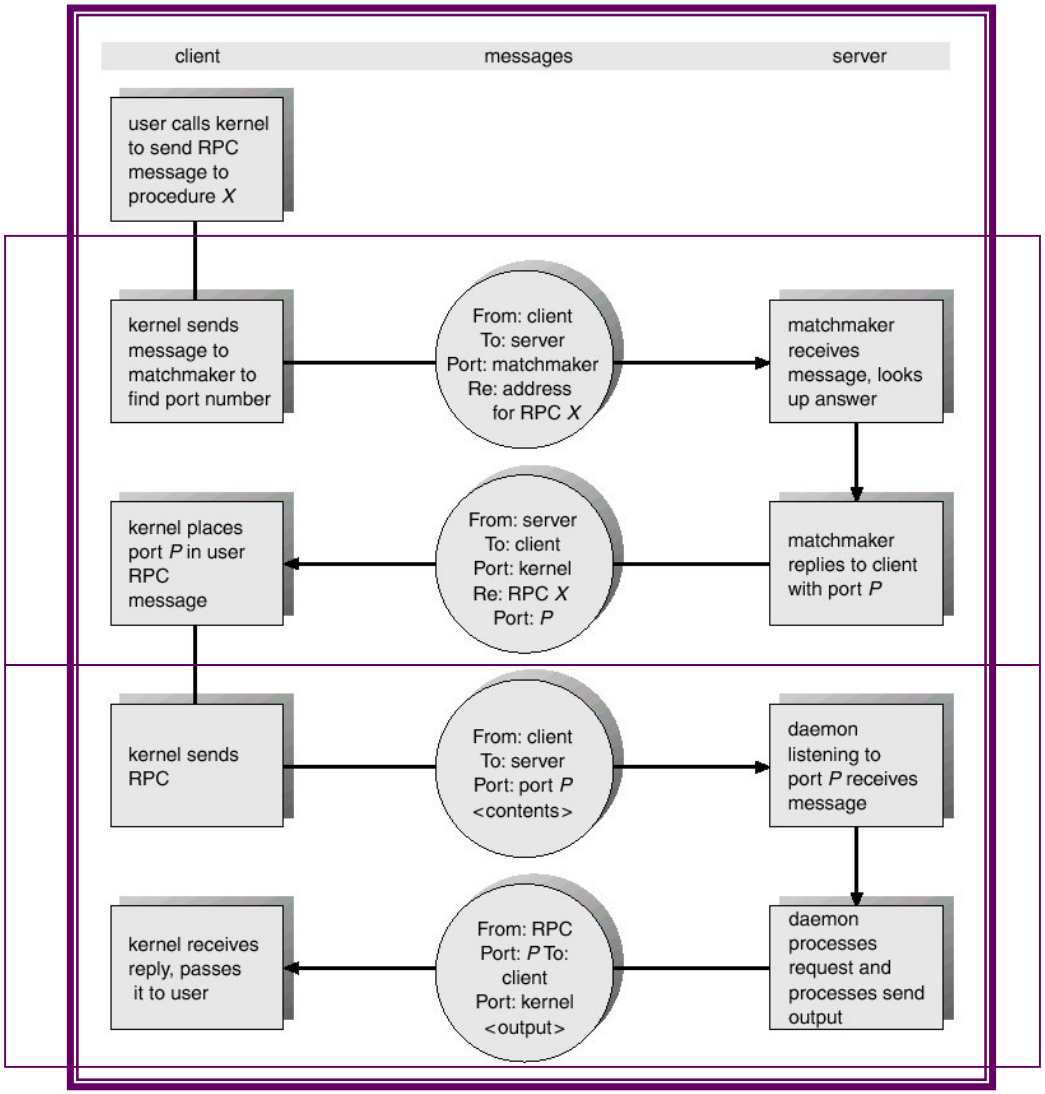
- Operation:

time	client	server
0		accept() blocked
1	new Socket()	read() blocked
2	write(1)	
3	read() blocked	s_in=1
4		write(2)
5	c_in=2	
6	exits	accept() blocked

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- *Stubs*:
  - Client-side proxy for the actual procedure on the server.
  - Server-side proxy listens to client-side proxy and executes actual procedure on server.
- Steps:
  - Client initiates RPC by calling client-side stub. Blocks.
  - The client-side stub locates the server and *marshalls* the parameters.
  - The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.
  - The server-side stub finishes by sending either procedure result or notification of completion to client-side stub.
  - Client-side stub returns. Client unblocks.

Execution of RPC

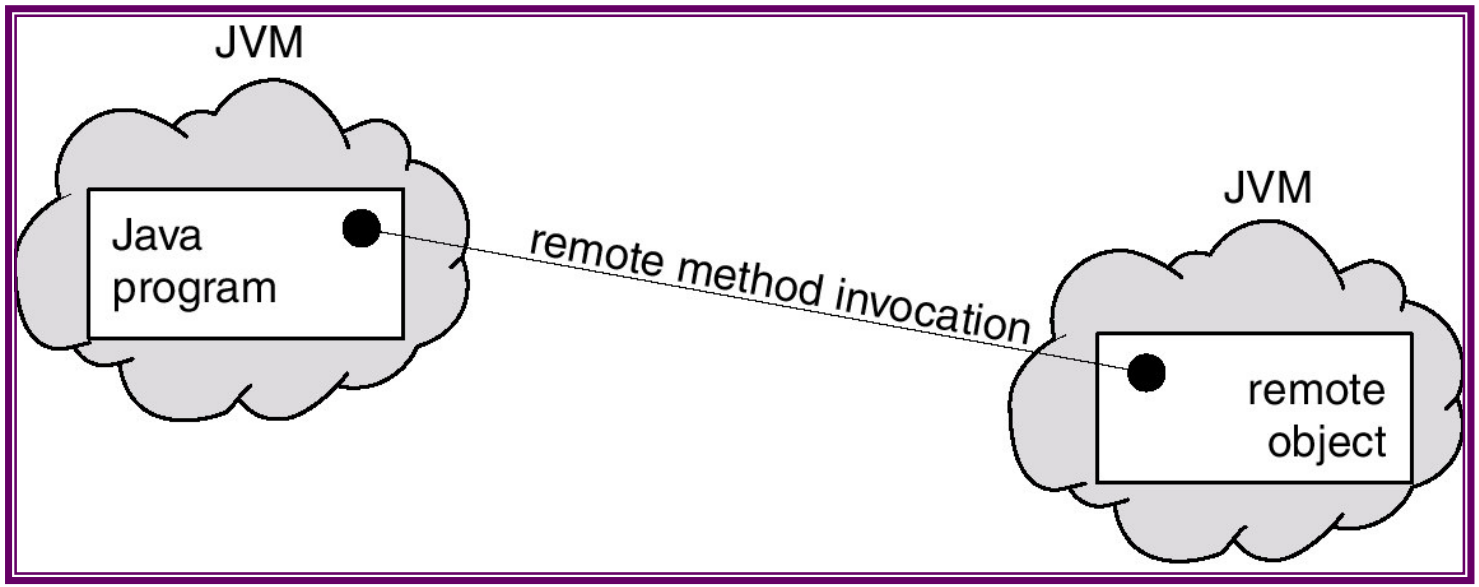


Discovery: find server containing procedure implementation

Execution: execute procedure passing arguments, return result

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.
- Java marshalls objects when local to client. Need compatible class definition.



Marshalling Parameters