Processes and Threads

Processes

- In most contemporary Operating Systems such as Windows and Linux/UNIX, the unit of management is called a **process**
- A process is a resource container
  - Depending on the specific operating system, a process will have a set of defining attributes
  - At any given moment, the collection of processes in a system completely defines the system
    - All computations must be done in the context of a process

Processes (cont'd)

- While processes on various systems share much more in common than in difference, we will focus on the process model used in **Linux**
- A Linux process is characterized by many attributes, but foremost among these are:
  - An executable program
  - One or more threads that can run the program
  - An address space to contain all process memory objects (i.e. text, data, stack, etc.)

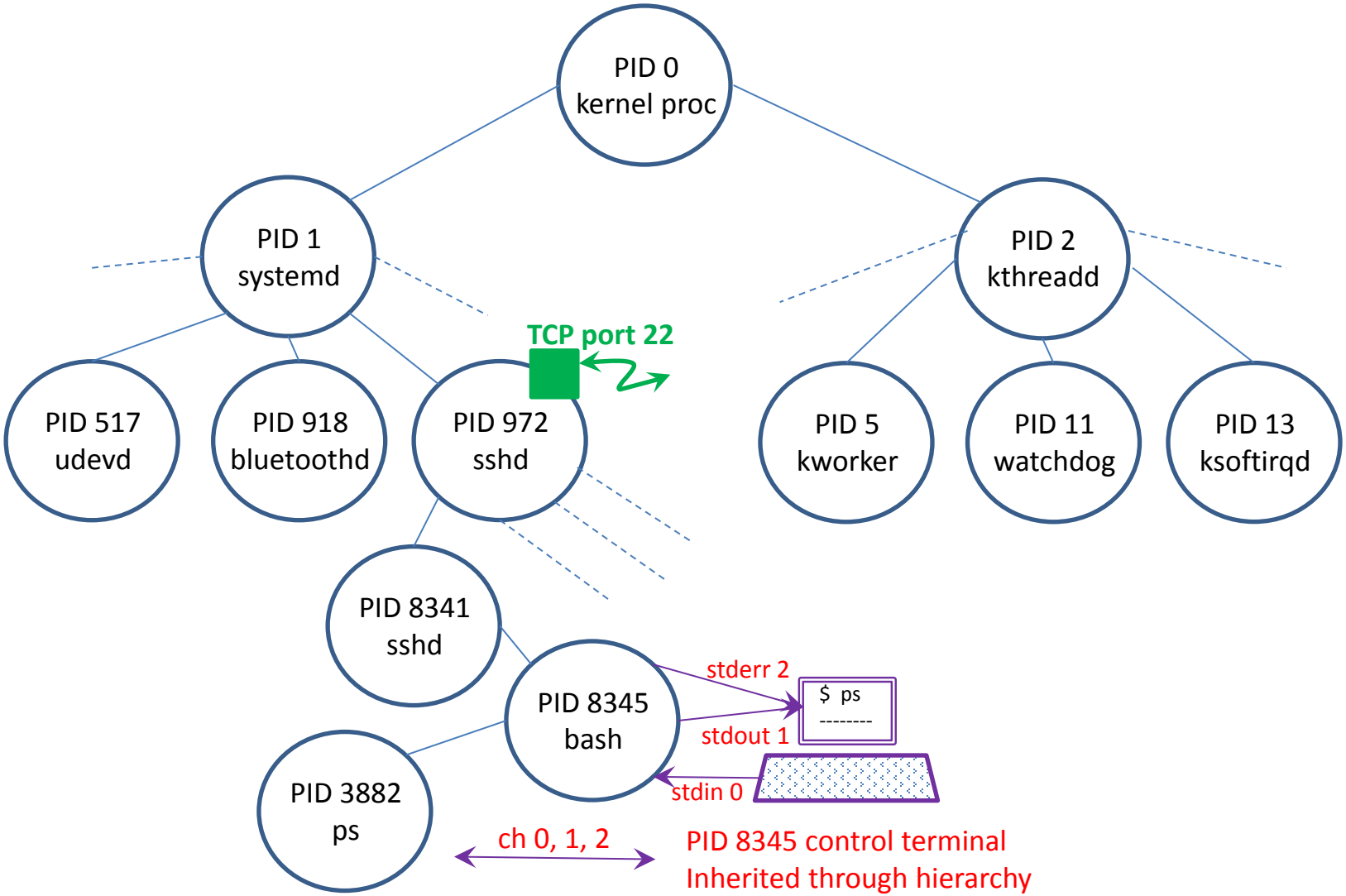
Processes (cont'd)

- The processes in a Linux system are identified by a simple integer value called a PID (Process Identifier)
  - During system boot, PID 0 is created, and it becomes the root of the ***process tree***
  - All subsequent PIDs are descended from PID 0
  - PIDs generally range from 0 to 32767 (32K - 1), and then roll back to unused smaller values
  - At any given instant, all in-use PIDs on a Linux system must be unique
  - When a process terminates, its PID value can be reused, but that won't happen until roll-over time

Processes (cont'd)

- PID 0 is a kernel process, and is not visible from user space
  - The kernel requires many housekeeping threads, and these are all owned by PID 0
- At boot time, PID 0 creates two children, PID 1 and PID 2
  - PID 1 runs a program named ***systemd*** (formally init)
  - PID 2 runs a program named ***kthreadd***
  - These daemon processes initialize the user space of a Linux system
  - All other processes in the system are directly descended from PIDs 1 or 2

A Linux Process Tree



Process Attributes

- Linux process attributes:
  - Executable program
    - A process must have an executable program loaded at all times (an executable ELF (Extended Link Format) file produced by the ld linker application)
    - A process can change its executable program if one of its threads makes the system call execve() or one of its derivatives (i.e. execl(), execlp(), etc.)
    - The exec…() family of system calls can load a new program into an existing process
      - The old address space defining code, data, stack, etc. is replaced by a new address space for the newly loaded program

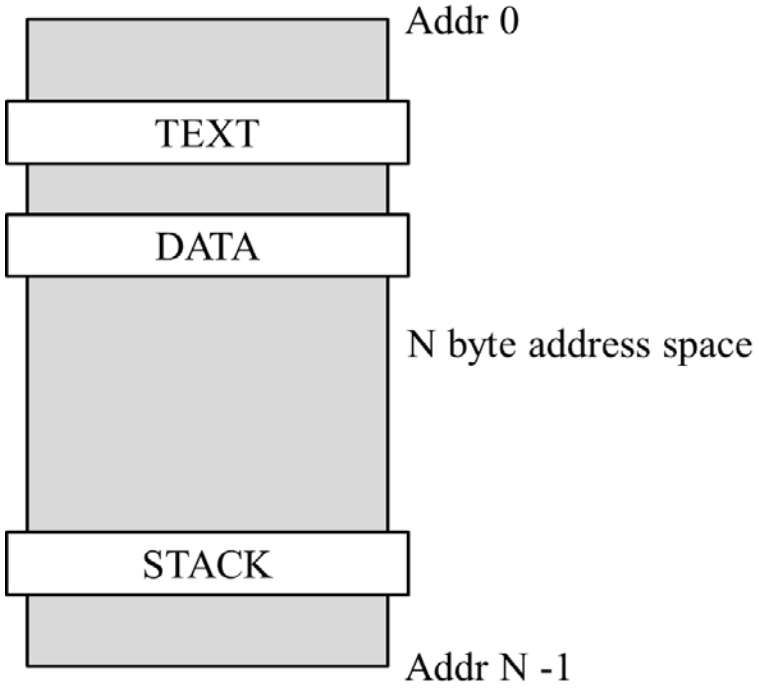
Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
    - The execution units (elements of code that can be scheduled to a processor) are called threads
    - Only threads can compute
    - If the last thread in a process terminates, then the process itself must terminate
    - All of the threads of a process share the common resources of the process, including the address space, control terminal, open files, signal behaviors, etc.

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
    - This is viewed as a contiguous collection of byte locations, some of which are populated by some form of contiguous memory object
    - An executing thread can access certain byte locations if such locations are part of a memory object, and if the thread has access permission to the memory object
    - Memory locations are accessed by either a ***load/store*** type data manipulation machine instruction (executed by a thread), or during the ***fetch instruction*** phase of the CPU's fetch-execute cycle

Process Address Space



- Each memory object is a contiguous range of bytes within the address space
- The size of the address space is limited by the CPU architecture and the operating system version
- In a 32 bit Linux system on an x86 processor, the user default space is 3 GB (it's 128 TB in a 64 bit x86 system)

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
    - A process cannot exist unless its address space is populated by at least the 3 memory objects shown
    - An address space may be populated by many more than just 3 objects
      - Text and data objects of attached shared libraries
      - Anonymous memory mapped heaps
      - Memory mapped files
      - Shared memory objects

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
    - A PID and PPID (parent PID)
    - User and Group IDs
      - RUID and RGID: the REAL user ID and group ID of the logged in user, as found in the user's login profile
      - EUID and EGID: the EFFECTIVE user ID and group ID of the process, used to determine what access rights a thread of this process has to other system objects (files, other processes, etc.)

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
  - Process default scheduling parameters
    - Typically inherited by the threads in a process
    - Scheduling POLICY, PRIORITY and AFFINITY
    - Each thread of a process can have its own specific policy, priority and affinity, but unless threads are created with custom values, they inherit the process defaults

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
  - Process default scheduling parameters
  - A working directory
    - The working directory of a process is shared by all threads in the process, providing a default location for file objects
    - If a thread uses a system call to change the working directory, all threads now see the new working directory

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
  - Process default scheduling parameters
  - A working directory
  - A channel table
    - Each process has a table to hold information about connections to external objects (devices, files, sockets, etc.)

Process Channel Table

Channel number (table index)	Close on Exec Flag (COE)	Open Object Pointer
0	No	stdin device
1	No	stdout device
2	No	stderr device
3	Yes	Socket connection
4	No	Unnamed pipe
5	No	Unnamed pipe
6	Yes	Ordinary file
7	No	Directory
Etc.	Yes/No	Other open objects
Etc.	…..	Free slots

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
  - Process default scheduling parameters
  - A working directory
  - A channel table
    - All external operations (read, write, control) require an open channel connection

Process Attributes (cont'd)

- Linux process attributes:
  - Executable program
  - One or more threads to run the program
  - An address space
  - Credentials
  - Process default scheduling parameters
  - A working directory
  - A channel table
  - A signal management table
    - Each process can control its behavior in the face of an asynchronous event called a signal

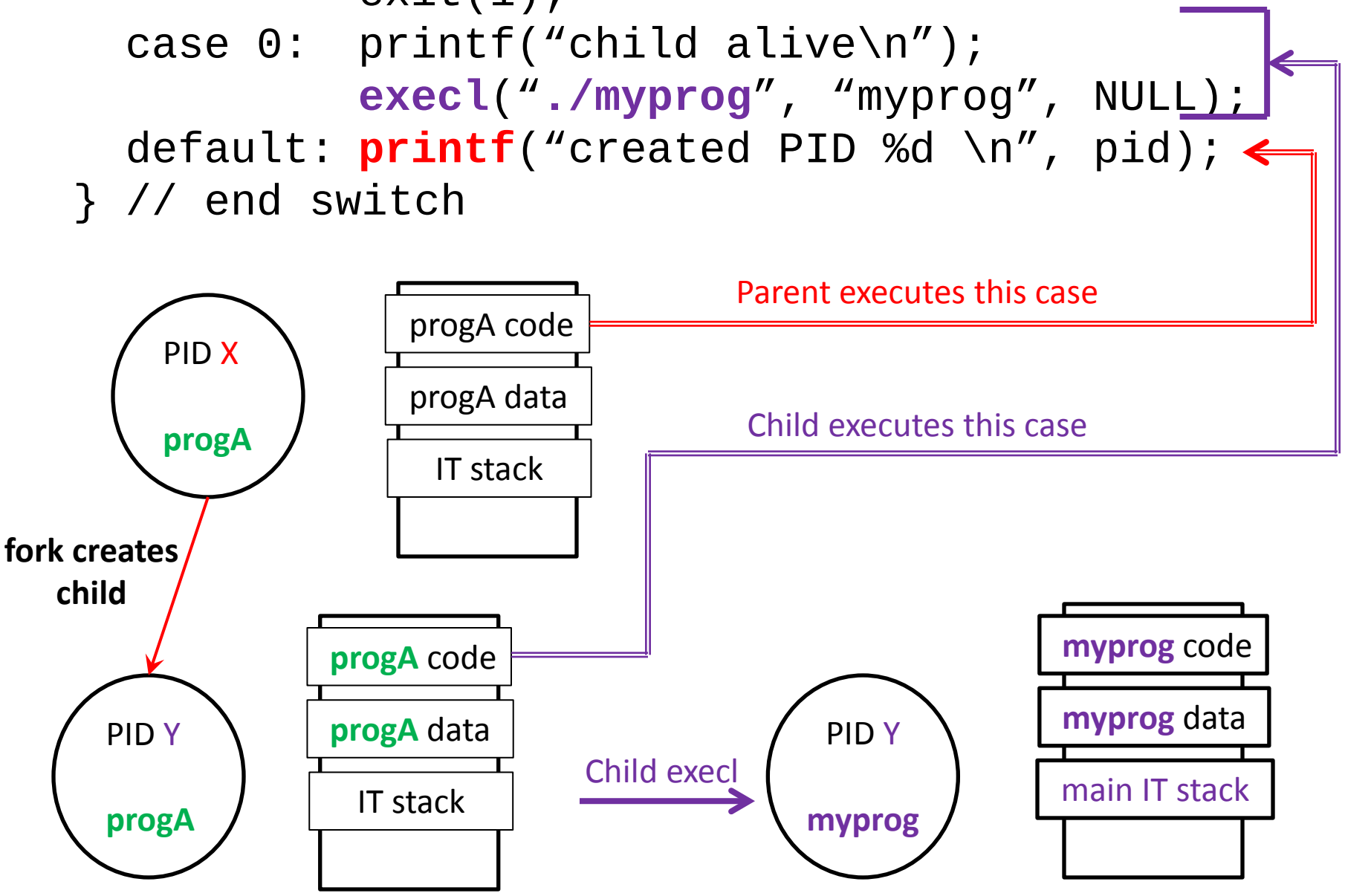
Process Signal Management Table

Signal number	Signal name	Signal disposition	Signal block mask	Signal flags
0	SIGNULL	SIG_DFL	0x0	0x0
1	SIGHUP	SIG_DFL	0x0	0x0
2	SIGINT	SIG_IGN	0x0	0x0
3	SIGQUIT	sig_funca()	0x35	0x0
…..				
11	SIGSEGV	sig_funcb()	0x737	0x0
…..				
15	SIGTERM	sig_funca()	0x35	SA_RESTART
…..				

Threads

- The executable (schedulable) elements in a Linux system
- Each thread in the system is uniquely contained by some process
  - Each user thread is contained by some user PID
  - Each kernel thread is contained in PID 0
- When a new process is created, it is populated by exactly one executable thread, known as the ***Initial Thread*** (IT) of the new process
- The IT of a process can create new threads only within its own process
- While the IT must create the ***second thread*** in a process, any subsequent threads can then create new threads, but only within their own process

```
switch (int pid = fork()){
  case -1: perror("fork failed ");
           exit(1);
  case 0:  printf("child alive\n");
           execl("./myprog", "myprog", NULL);
  default: printf("created PID %d \n", pid);
} // end switch
```

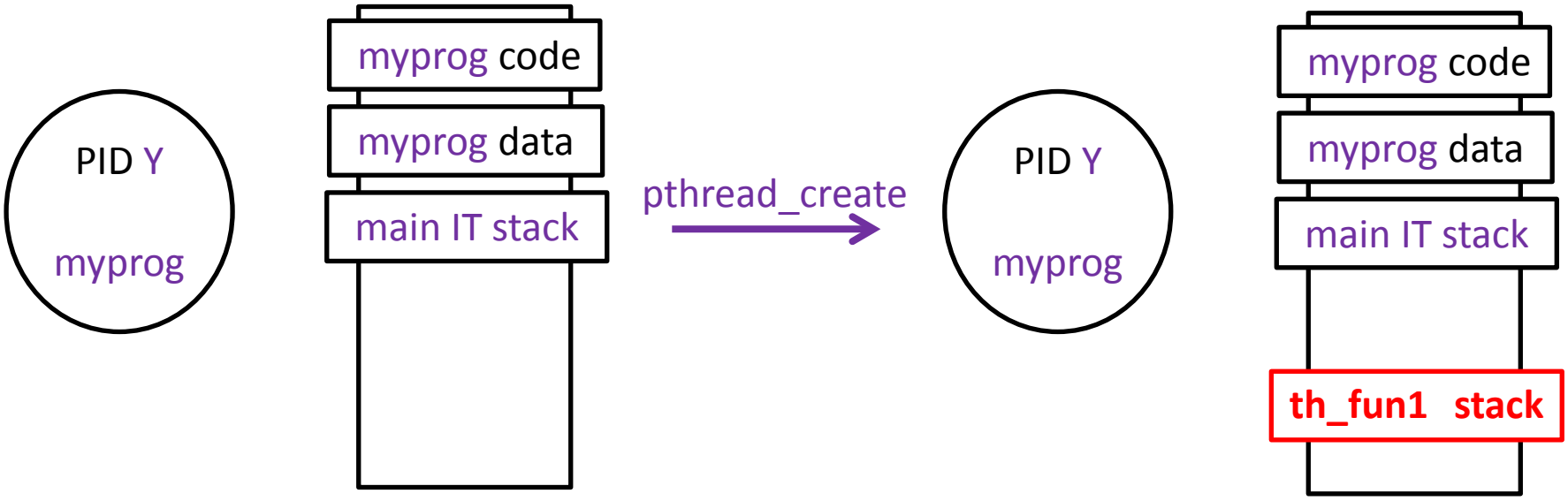


The new child program **myprog** executes from the first statement in its **main()** function.

If the new program executes the following statement:

```
pthread_create(&tid_id, NULL, th_fun1, NULL);
```

a **new stack** will be mapped into the address space



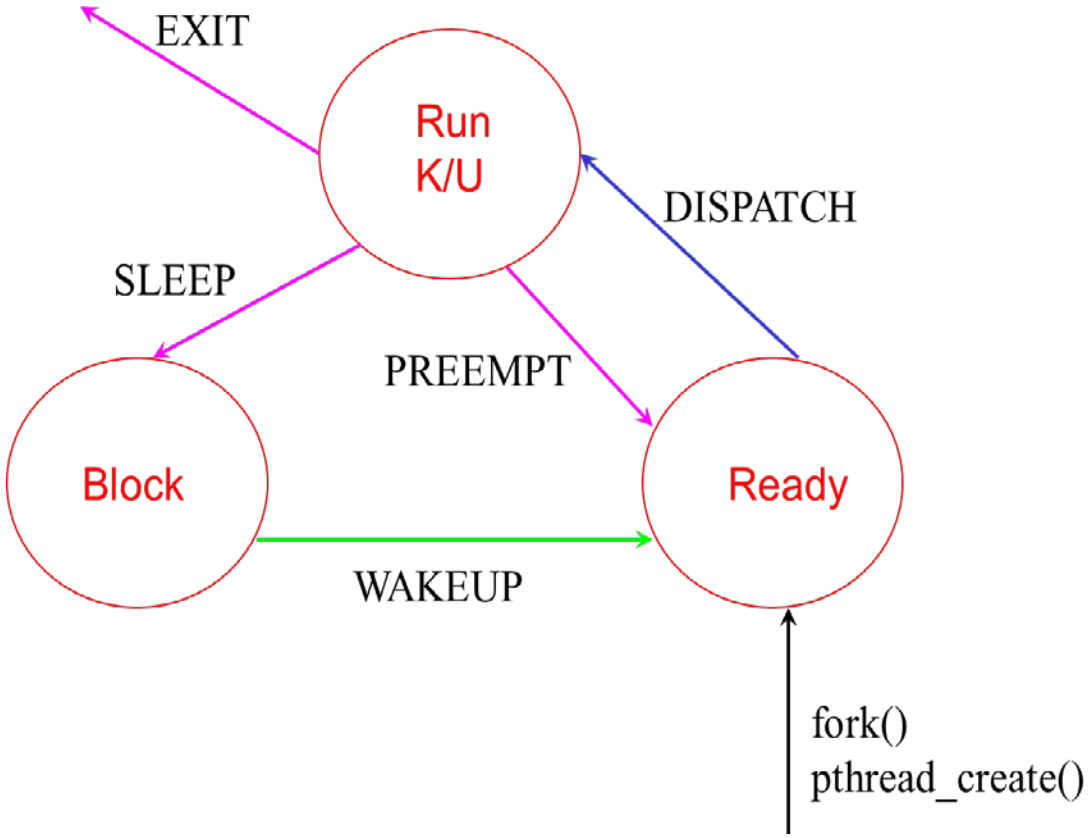
Thread Attributes

- A thread in a Linux system has the following attributes:
  - A hardware state
    - Because threads are runnable entities, the kernel must initialize and save their registers and CPU state
    - Whenever a thread undergoes a context switch, its hardware state is either saved or restored
    - Each thread has a private hardware state, but all threads share the hardware state of the process they are contained in (the address space)

Thread Attributes (cont'd)

- A thread in a Linux system has the following attributes:
  - A hardware state
  - A software state
    - An existing thread is always in one of three basic software states
      - Running in either ***kernel*** or ***user*** mode
      - Blocked and waiting for some ***event*** to occur that will cause it to transition to the ***ready state***
      - Ready to run and waiting for a ***CPU*** to run on and so transition to the ***running*** state when the CPU becomes available

Thread States and Transitions



Thread Attributes (cont'd)

- A thread in a Linux system has the following attributes:
  - A hardware state
  - A software state
  - A signal state
    - Each thread can selectively block certain synchronous signals (SIGSEGV, SIGFPE, etc.)
    - All threads in a process share the same signal disposition for any given signal
      - Threads can choose to selectively block a given signal, but if they allow a signal to be delivered, then the behavior will be that of the process disposition for that signal (i.e. default, ignore, handle)

Thread Access to Objects

- Since only threads can execute, only threads can attempt to access a system object
- All system objects are labeled with an owner UID and a group GID
- System objects include
  - Objects controlled by open channels (ordinary files, directories, symbolic links, devices, pipes and sockets)
  - Processes and threads
  - System V IPC (inter-process communication) objects (semaphores, shared memory segments, message queues)

Thread Access to Objects (cont'd)

- When a thread attempts to access an object, the ***credentials*** of the process that the thread is contained in determine success or failure
- All processes have an EUID and an EGID that the kernel uses to determine if a thread can access a given object
  - An attempted operation by a thread in one process against another process requires a UID match (e.g. sending a signal to another PID)
  - Certain control operations on file and IPC objects also require a UID match (e.g. using chmod to change permissions on a file object)
  - General operations on file and IPC objects (e.g. read/write data) require permission matching based in EUID and EGID

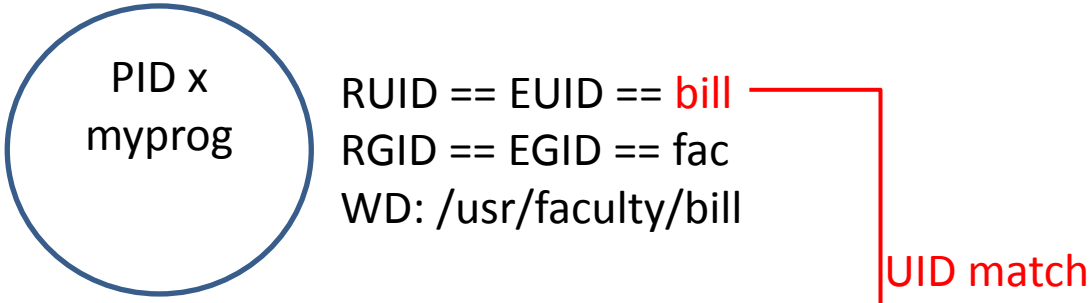
Thread Access to Objects (cont'd)

- File and IPC objects are represented by controlling data structures that contain permission bits for the object
- Permission bits are organized into three categories
  - Owner permissions
  - Group permissions
  - Other permissions
- Each category normally includes 3 bits called the read, write and execute bits

Thread Access to Objects (cont'd)

- General operations on file and IPC objects always begin by a thread attempting to create/open or just open such an object
- The kernel considers the credentials of the requesting thread (EUID and EGID) to determine if the initial access can succeed
- The kernel always performs the following checks:
  - If the EUID of the calling thread matches the UID of the target object, then decide using RWX owner bits only
  - If there is no UID match, but the EGID of the caller matches the GID of the target, then decide using RWX group bits only
  - If both UID and GID matches fail, then decide using RWX other bits only
  - Only three bits are ever used to make the decision

Thread Access Example




```
[bill@mich-fc17-126 bill]$  pwd
/usr/faculty/bill
[bill@mich-fc17-126 bill]$  ls -l
total 48
-rw-r--r--. 1 bill fac   20483 Aug  6 11:33 mmap_rw_var.c
-r--rw-rw-. 1 bill fac   22097 Aug  6 11:34 malloc_th.c
```

- A ***system call*** made by a thread in PID x is:

  ```
  int channel = open("/usr/faculty/bill/mmap_rw_var.c", O_RDWR, 0);
  ```

- The system call ***succeeds*** and returns a valid channel to read and write
- A second call made by a thread in PID x is:

  ```
  int channel = open("/usr/faculty/bill/malloc_th.c", O_RDWR, 0);
  ```

- This call ***fails***, since the calling process is the owner, and owner permissions don't allow WRITE, even though group and other do

Thread Synchronization Requirements

- A thread may find itself running on one core of a system while another thread is ***concurrently*** running on a different core
  - Both threads are computing on their own hardware resources (separate registers, program counters, etc.)
  - If these threads share common data (located in RAM), then they must manipulate that data ***atomically*** to avoid data corruption
  - Sections of contending code of this type are referred to as ***critical sections***, and require synchronization

Thread Synchronization (cont'd)

- In reality, any operating system that enables ***timesharing*** execution faces the same thread synchronization needs even when only a ***single core*** is involved. Here threads can run in ***virtual concurrency***
  - If thread is in the middle of modifying a shared structure, for example, the thread may lose its core to a contending thread due to a time-slice expiration
  - The contending thread may gain the core and begin working on the partially modified structure (the timed-out thread hasn't finished), leading to a corrupt result
  - Even though there is only one set of hardware resources here, the outbound thread stores its registers for later use and the inbound thread reloads the registers for itself

Thread Synchronization (cont'd)

- To update a value stored in RAM, a thread must
  1. Load the value into a core register
  2. Perform some sort of modification operation
  3. Store the value back to RAM
- If one thread carries out the first 2 steps, but a second thread loads the original value from RAM before the first thread completes step 3, then the value loaded by the second thread is logically corrupt
- The order of simple memory reference operations is stochastic among multiple cores

Thread Synchronization (cont'd)

- A synchronization mechanism that solves the critical section problem must have the following attributes:
  - Contending threads must be able to synchronize ***regardless of the speeds*** their respective cores operate at
  - Synchronization requires ***mutual exclusion***
  - Mechanisms must support ***progress***
  - Mechanisms must support ***bounded waiting***

Synchronization Mechanisms

- Software solutions
  - Peterson's algorithm (see: http://www.cs.uml.edu/~bill/cs515/Petersons_Sync_Alg.txt)
- Hardware solutions
  - Interrupt disable (single core systems)
  - Spin lock instruction support (multicore systems)
    - Intel: XCHG Rx, mem_adr
    - Once executed, guarantees that register Rx is exchanged atomically with RAM at mem_adr
    - This allows the three steps (previously shown) in a memory modification to be done in a single instruction

Spin Lock with Peterson's Algorithm

```
#define         FALSE   0
#define         TRUE    1
#define         N       2  // number of processes

int   turn;               // whose turn is it?
int   interested[N];      // all values initially 0 (FALSE)

void enter_region(int process)  // process is 0 or 1
{
  int other;                    // other process number
  other = 1 - process;          // the opposite of process
  interested[process] = TRUE;   // raise my interested flag
  turn = process;               // set turn to yourself
  while(interested[other] == TRUE && turn == process); // spin
}

// call enter_region(process_value) where process_value is 1 or 0
// execute critical section
// call leave_region(process_value) where process_value is 1 or 0

void leave_region(int process)  // process is 0 or 1
{
  interested[process] = FALSE;  // drop my interested flag now
}
```

Spin Lock with XCHG

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Synchronization Foundations

- A software spin lock does not require any special hardware instructions, but is very inefficient
- Hardware support for spin locks like the Intel special XCHG instruction, allows efficient implementations
- In either case, spin locks serve as the foundation for all other forms of synchronization mechanisms

Blocking Synchronization Mechanisms

- Spin locks are reasonable mechanisms when the contention for a critical section is light, and when the content of a critical section is short and succinct
- When a thread faces an extended period of spinning when waiting for its critical section, spin locks may not be a reasonable mechanism to use
- Blocking lock mechanisms allow a waiting thread to move to the block state and give up its core to some other productive thread while waiting

Blocking Mechanisms (cont'd)

- We will consider 4 blocking mechanisms:
  - Semaphores
  - Event Counters and Sequencers
  - Pthread Mutex Locks
  - Pthread Condition Variables
- There are many others, some of which are discussed in the book
- They are all implemented on top of some spin lock mechanism

Semaphores

- Semaphores are constructs that logically consist of a counter and a waiting queue
- The basic functions on a semaphore include a conditional decrement operation known by one of the names
  - ***wait(sem); p(sem); down(sem);***

  and a conditional increment operations known by one of the names:
  - ***signal(sem); v(sem); up(sem);***

Semaphores (cont'd)

- A ***wait(sem)*** operation is an attempt by the caller to ***decrement*** the semaphore counter by 1
  - If the counter is non-zero, the call will succeed and **return immediately** to the called, leaving the semaphore counter decremented by 1
  - If the value of the semaphore counter is already zero, then the calling thread will be placed on the semaphore ***block queue***, and is forced to yield its core to some other thread
    - The blocked thread may be moved to the ready state sometime in the future when a ***signal(sem)*** call is made

Semaphores (cont'd)

- A ***signal(sem)*** operation is an attempt by the caller to ***increment*** the semaphore counter by 1
  - The call will always succeed and **return** to the caller
  - If the counter is **non-zero**, then there are no waiters, and the calling thread simply increments the semaphore counter by 1 and returns
  - If the value of the semaphore counter is **zero**, then the calling thread will check to see if there are any waiters
    - If one or more waiters are on the block queue, the calling thread will move one of them to the ready state and then return
    - If there are no waiting threads on the block queue, the calling thread will simply increment the semaphore count by 1 and return

A single producer, single consumer ring buffer, synchronized with **counting** semaphores

GLOBAL TO PRODUCER AND CONSUMER THREADS:

```
sem_t  prod = 10;  // semaphore initialized  to 10 spaces
sem_t  cons = 0;   // semaphore initialized  to 0 objects

int buf[10], in=0, out=0;  // 10 element ring buffer and pointers
void p ( sem_t  * );       // available p() and v() functions
void v ( sem_t  * );
```

PRODUCER FUNCTION

```
void producer(){
while(1){
  p(&prod);
  buf[in] = random();
  in = (in + 1) % 10;
  v(&cons);
}
```

CONSUMER FUNCTION

```
void consumer(){
int val;
while(1){
  p(&cons);
  val = buf[out];
  // print val somewhere
  out = (out + 1) % 10;
  v(&prod);
}
```

A multi producer, multi consumer ring buffer, synchronized with **counting** semaphores and **binary** semaphores

GLOBAL TO ALL PRODUCERS AND CONSUMER THREADS:

```
sem_t  prod = 10, inlock = 1;   // binary semaphores set to unlock
sem_t  cons = 0, outlock = 1;
```

```
int buf[10], in=0, out=0;  // 10 element ring buffer and pointers
void p ( sem_t  * );       // available p() and v() functions
void v ( sem_t  * );
```

PRODUCER FUNCTION

```
void producer(){
while(1){
   p(&prod);
   p(&inlock);
   buf[in] = random();
   in = (in + 1) % 10;
   v(&inlock);
   v(&cons);
}
```

CONSUMER FUNCTION

```
void consumer(){
int val;
while(1){
   p(&cons);
   p(&outlock);
   val = buf[out];
   // print val somewhere
   out = (out + 1) % 10;
   v(&outlock);
   v(&prod);
}
```

Weak Reader Preference Solution to the Reader-Writer Problem Using Semaphores

GLOBAL TO ALL READER AND WRITER THREADS:

```
sem_t  rdlock = 1 , wrlock = 1;   // binary semaphores set to unlock
int  nreaders = 0;                // count of current readers
```

READER FUNCTIONS

```
void reader_in(){
  p(&rdlock);
  if(nreaders == 0)
    p(&wrlock);
  ++ nreaders;
  v(&rdlock);
} // reader can read

void reader_out(){
 p(&rdlock);
 --nreaders;
 if(nreaders == 0)
    v(&wrlock);
 v(&rdlock);
}
```

WRITER FUNCTION

```
void writer(){
 p(&wrlock);
 // writer can write
 v(&wrlock);
}
```

Semaphores (cont'd)

- The semaphore counter and its block queue require atomic access to avoid corruption
- Each semaphore requires a spin lock mechanism to protect its counter and queue
- This is reasonable application of a spin lock, since we hold such a lock for just a short period of time to check/update the counter and possibly queue or de-queue a thread
- Spin locks always provide the foundation for blocking locks

Event Counters and Sequencers

- While semaphores can solve all complex total ordering synchronization problems, for certain partial order problems they are more complex than what is required
- Semaphore can be logically decomposed into an event counter mechanism and a sequencer mechanism
- Partial order problems can be solved very efficiently using only event counters
- Total order problems can be solved using a combination of event counters and sequencers, in a fashion similar to semaphores
- Event counters also have a broadcast facility, that is not found with semaphores

Event Counters

- Event counters are unsigned, monotonic counters that are always initialized to zero, and a waiting queue (as for a semaphore)
- The basic functions on an event counter include:
  - ***await(EC, uint value);***
  - ***advance(EC);***
- A thread that calls **await()** may return immediately or it may block and lose its core
- A thread that calls **advance()** always returns immediately

Event Counters (cont'd)

- The ***await(EC, uint value)*** call has the following semantics:
  - if **value > EC** then block and yield core;
    else return immediately;
- So, if the event you're interested in hasn't happened yet, block and wait for it, otherwise return right away:
  - If we assume that an EC is currency set to 42 and a thread calls **await(EC, 50);** the calling thread will block on the waiting queue, since it can't proceed until the 50^{th} event occurs and only 42 events have occurred so far
  - A call of **await(EC, 40)** however, would return immediately, since the caller wants to proceed after the 40^{th} event, and 42 events have already occurred

Event Counters (cont'd)

- The ***advance(EC)*** call has the following semantics:
  - Increment the EC
  - If the new value of the **EC >= any value** that any thread has called await() with, then move all such blocked threads to the **ready state**
- So, if a thread advances an EC, it must check for any blocked threads associated with this EC, and if the EC value is now equal to or grater than any thread's waiting value, the calling thread must move such threads to the ready state
  - If we assume that an EC is currently set to 42 and a thread calls **advance(EC);** the calling thread will increment the EC to 43
    - Any thread(s) waiting on this EC for a value of 43 or less will be moved to the **ready state**

A single producer, single consumer ring buffer, synchronized with **event counters only**

GLOBAL TO PRODUCER AND CONSUMER THREADS:

```
ec_t pEC, cEC;  // event counters always init 0
int ring_buf[10]; // 10 element ring buffer
void await (ec_t *, int); // ec operations
void advance    (ec_t *);
```

PRODUCER FUNCTION

```
void producer(){
unsigned in = 0;
while(1){
  await(&pEC, in – 10 + 1);
  ring_buf[in % 10] = random();
  in = (in + 1);
  advance(&cEC)
}
```

CONSUMER FUNCTION

```
void consumer(){
int val;
unsigned out;
while(1){
  await(&cEC, out + 1);
  val = ring_buf[out % 10];
  // print val somewhere
  out = (out + 1);
  advance(&pEC);
}
```

Sequencers

- When total ordering problems must be synchronized, the addition of the sequencer mechanism with event counters provides all of the power of a semaphore
- A sequencer is an **atomic counter** with a single operation:
  - ***uint my_tix = ticket(seq);***
- Sequencers are always **initialized to zero**, and will return zero to the first thread that makes the ***ticket()*** call, while subsequent calls will return monotonically increasing unsigned integers (ie. 1, 2, 3, … , etc.)

A multi producer, multi consumer ring buffer, synchronized with **event counters and sequencers**

GLOBAL TO PRODUCER AND CONSUMER THREADS:

```
ec_t pEC, cEC;  // event counters always init 0
seq_t pSEQ, cSEQ // sequencers always init 0
int ring_buf[10]; // 10 element ring buffer
void await (ec_t *, int); // ec operations
void advance    (ec_t *);
unsigned ticket(seq_t *); // sequencer operation
```

PRODUCER FUNCTION

```
void producer(){
unsigned t;
while(1){
  t = ticket(&pSEQ);
  await(&cEC, t);
  await(&pEC, t – 10 + 1);
  ring_buf[t % 10] = random();
  advance(&cEC)
}
```

CONSUMER FUNCTION

```
void consumer(){
int val;
unsigned u;
while(1){
  u = ticket(&cSEQ);
  await(&pEC, u);
  await(&cEC, u + 1);
  val = ring_buf[u % 10];
  // print val somewhere
  advance(&pEC);
}
```

Weak Reader Preference Solution to the Reader-Writer Problem Using Evencounters and Sequencers

```
ec_t    rdE,    wrE;        // event counters always init 0
seq_t   rdS,    wrS         // sequencers always init 0
int  nreaders = 0;          // count of current readers
```

READER FUNCTIONS

```
void reader_in(){
  await(&rdE, ticket(&rdS));
  if(nreaders == 0)
   await(&wrE, ticket(&wrS));
  ++ nreaders;
  advance(&rdE);
} // reader can read

void reader_out(){
 await(&rdE, ticket(&rdS));
 --nreaders;
 if(nreaders == 0)
    advance(&wrE);
 advance(&rdE);
}
```

WRITER FUNCTION

```
void writer(){
await(&wrE, ticket(&wrS));
 // writer can write
 advance(&wrE);
}
```

IEEE pthread API

- The POSIX pthread API (avavilable on virtually all UNIX-based systems) provides various functions to create and control threads within a process
- When a process is populated by more than one thread, the co-existing threads may require synchronization when manipulating global variables
- Since each thread runs on its own stack, local variables generally do not require synchronization, but global variables are visible to all threads within a given process, and may be corrupted if multiple threads attempt (virtually or physically) concurrent updates

pthread Mutex Locks

- Mutual exclusion lock support is provided by the pthread library (see pthread.h)
  - **pthread_mutex_t** // type of a mutex lock
  - ***pthread_mutex_lock(pthread_mutex_t *)***
  - ***pthread_mutex_unlock(pthread_mutex_t *)***
- Mutex locks are similar to **binary semaphores**, but are much more efficient when used within a single process
  - A **mutex** does **NOT** have an associated counter, and so cannot be used like a counting semaphore
  - When the ***pthread_mutex_lock()*** routine returns to a calling thread, that thread is guaranteed to be the only holder of that lock until the thread releases it with a corresponding ***pthread_mutex_unlock()*** call

pthread Mutex Locks (cont'd)

- For certain update synchronization, a mutex alone provides sufficient functionality
  - To carry out the updating of some data structure, for example, we can associate a mutex with the structure, and require a thread to obtain the mutex before proceeding with the update.
    - This works fine if we want to atomically increment a counter, for example
  - As long as we can be sure that our update intentions can be satisfied once we obtain the lock, then a mutex is all we need

The "Observer – Reporter" problem: an observer thread and a reporter thread must keep their counter use coherent using a single pthread mutex

GLOBALS TO OBSERVER AND REPORTER:

```
pthread_mutex_t     obj_lock;
int                 obj_count  = 0;
int  pthread_mutex_lock(pthread_mutex_t   *mlock);
int  pthread_mutex_unlock(pthread_mutex_t   *mlock);
```

OBSERVER THREAD FUNCTION

```
void *observer (void * arg){
  while (1){
    // when object passes
    pthread_mutex_lock(&obj_lock);
    ++obj_count ;  // increment counter
    pthread_mutex_unlock(&obj_lock);
  }   // while
}     // observer
```

REPORTER THREAD FUNCTION

```
void *reporter (void * arg){
 while (1){
   sleep (900);  // sleep for 15 min
   pthread_mutex_lock(&obj_lock);
   // report obj_count in some way
   obj_count = 0;       // reset counter
   pthread_mutex_unlock(&obj_lock);
  }   // while
}     // reporter
```

pthread Mutex Locks (cont'd)

- Under certain conditions, however, we may not be able to proceed once we obtain a mutex
  - Consider a ring buffer
    - We have an integer variable that keeps the space count
    - We have a mutex to protect this variable
    - As a producer or consumer, we must obtain the mutex before we can manipulate the space count
    - Assume that a producer obtains the mutex, and when checking the space count finds that it is zero
      - The producer cannot produce since there is no slot to produce into
      - The space count cannot now change, until the mutex is available (in this case a consumer who clears out a space needs to increment the space count, but is not allowed to update it until it obtains the mutex)
      - If the producer unlocks the mutex, what should it do after that ?

pthread Condition Variables

- Because a mutex alone is insufficient to solve certain types of problems, the pthread API also includes a type of object called a **condition variable**
  - **pthread_cond_t**
  - ***pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *)***
  - ***pthread_cond_signal(pthread_cond_t *)***
- A condition variable allows a thread to block itself if the conditions it needs to proceed are not currently available
  - A producer, for example, that finds that there are no available slots in a ring buffer to produce into
  - The ***pthread_cond_wait()*** call will place the calling thread into the waiting queue of the condition variable given in the **first argument** (allowing some other thread to use its core)
    - The call also releases the **mutex** referenced in the **second argument**

pthread Condition Variables (cont'd)

- If a thread is placed in the waiting queue of a condition variable, it is logically waiting to return from its call to ***pthread_cond_wait()***
  - When blocked, its **mutex** argument is unlocked
  - Some other thread can now obtain this **mutex**
  - The thread now holding the **mutex**, can update a dependent variable (like the space count)
  - Once updated, the **mutex** can be unlocked
    - The updating thread (a **consumer** for example) now has an obligation to use the ***pthread_cond_signal()*** function to bring the queued **producer** back to the ready state

pthread Condition Variables (cont'd)

- If a thread is blocked on a condition variable and that variable is signaled, the blocked thread will be made ready and have a chance to execute again
  - The blocked thread can only be made ready if the condition variable has been signaled **AND** if the dependent **mutex** is unlocked
  - The thread continues execution on the return of the ***pthread_cond_wait()*** call that put it to sleep
    - Upon return, the thread is guaranteed to be the sole possessor of the dependent **mutex**
    - This allows the thread to safely manipulate any variables (like free slot count) that the **mutex** logically protects

pthread Condition Variables (cont'd)

- When a thread returns from a ***pthread_cond_wait(),*** it owns the associated **mutex** and can safely manipulate any protected variables this **mutex** guards
  - There is **NO** guarantee that such variables are in the condition the thread may need them to be in
    - An awaking producer thread for example, may still find that the space count is zero
    - Since a thread must assume that it can be awakened at any time from a condition wait, it is critical for the waking thread to recheck its conditions before it proceeds
    - A ***pthread_cond_wait()*** call should always be place in a **while loop**

A multi producer, multi consumer ring buffer, synchronized with **pthread mutexes and condition variables**

GLOBAL TO PRODUCER AND CONSUMER THREADS:

```
pthread_mutex_t Plock, Clock; // mutex locks
pthread_cond_t Pcond, Ccond;  // condition vars
int ring_buf[10]; // 10 element ring buffer
int space = 10, objects = 0, in = 0, out = 0; // controls
pthread_cond_wait(pthread_cond_t *condx, pthread_mutex_t *mlk);
pthread_cond_signal(pthread_cond_t *condx);
```

```
void producer(){
while(1){
 pthread_mutex_lock(&Plock);
 while(space == 0)
  pthread_cond_wait(&Pcond,&Plock);
 ring_buf[in % 10] = random();
 ++in, --space;
 pthread_mutex_unlock(&Plock);

pthread_mutex_lock(&Clock);
 ++objects;
 pthread_mutex_unlock(&Clock);
 pthread_cond_signal(&Ccond);
}
```

```
void consumer(){
int val;
while(1){
 pthread_mutex_lock(&Clock);
 while(objects == 0)
  pthread_cond_wait(&Ccond,&Clock);
 val = ring_buf[out % 10];
 // do something with val
 ++out, --objects;
 pthread_mutex_unlock(&Clock);
 pthread_mutex_lock(&Plock);
 ++space;
 pthread_mutex_unlock(&Plock);
 pthread_cond_signal(&Pcond);
}
```

Weak Reader Preference Solution to the Reader-Writer Problem Using pthread mutexes

GLOBAL TO READER AND WRITER THREADS:

```
pthread_mutex_t rdlock, wrlock; // mutex locks
int nreaders = 0;               // number of readers
```

READER FUNCTIONS

```
void reader_in(){
 pthread_mutex_lock(&rdlock);
 if(nreaders == 0)
    pthread_mutex_lock(&wrlock);
 ++ nreaders;
 pthread_mutex_unlock(&rdlock);
} // reader can read

void reader_out(){
 pthread_mutex_lock(&rdlock);
 -- nreaders;
 if(nreaders == 0)
    pthread_mutex_unlock(&wrlock);
 pthread_mutex_unlock(&rdlock);
}
```

WRITER FUNCTION

```
void writer(){
 pthread_mutex_lock(&wrlock);
 // writer can write
 pthread_mutex_unlock(&wrlock);
}
```

Thread Scheduling

- Since threads are the runnable elements of a process, how threads are placed on execution cores is a major issue to an operating system
- Scheduling policies
  - A scheduling policy is a rubric for determining how threads are dispatched to a core
  - There are many different policy designations, but generally there are 2 basic policy categories
    - Timesharing
    - Real-Time

Scheduling Policies

- Timesharing (default in Linux)
  - A timesharing thread is created with
    - A priority
    - An execution time slice (quantum)
    - A list of cores upon which it can run (affinity attributes)
  - The specific priority and quantum of such a thread may be dynamically adjusted during execution
    - The system typically collects some execution history and may adjust these attributes accordingly
      - This adjustment procedure is often called aging
      - A compute-bound thread may have its priority lowered
      - An interactive thread may have its priority increased
      - In Linux, this policy is formally known as SCHED_NORMAL

Scheduling Policies (cont'd)

- Real-Time
  - A real-time thread is created with
    - A priority
    - Possibly an execution time slice (quantum)
    - A list of cores upon which it can run (affinity attributes)
  - The specific priority and quantum (if appropriate) of such a thread **IS NOT** dynamically adjusted
    - The operating system typically has a HANDS-OFF strategy
      - Real-time thread attributes may be manipulated programmatically, but the system leaves them alone
      - The real-time policies of Linux (POSIX) are called SCHED_FIFO (no quantum used) and SCHED_RR (quantum used with Round-Robin)

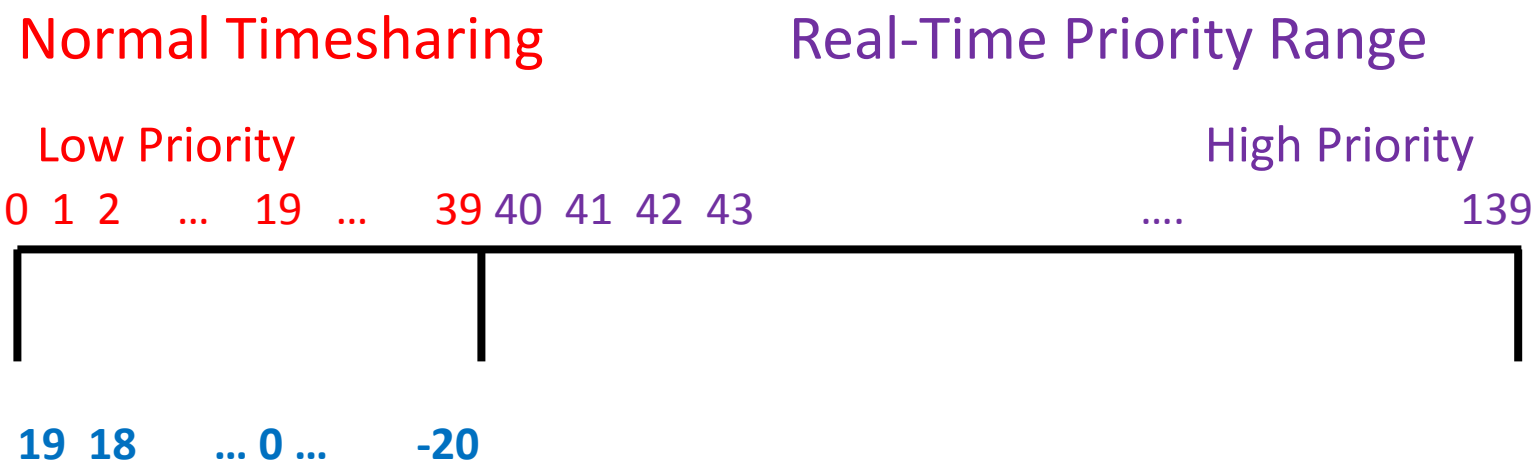
Scheduling Priorities

- Priorities are typically organized by policies
  - In Linux, any normal thread will always have a lower priority than any real-time thread
  - Scheduling algorithms typically seek the highest priority thread available to satisfy a context switch operation (Highest Priority First, HPF)
  - In Linux, if a real-time thread is in the READY state during a context switch, it will always be dispatched before any normal thread
  - Most systems, like Linux and Windows, have a contiguous range of priority values, partitioned by normal and real-time policies

Linux Scheduling

- Linux uses a contiguous range of priority values from 0 to 139 (140 scheduling queues)
  - The values from 0 to 39 are used for normal timesharing threads (0 is low priority, 39 is high)
  - Values from 40 to 139 are used by real-time threads (which consider their own range as 0 – 99)
  - So a thread running with a priority of 20 is a normal timesharing thread while one with a priority of 53 is a real-time thread
  - When a context switch is imminent on a core, the scheduler finds the highest priority populated queue (of the 140 queues) and dispatches the thread at the head of this queue (with or without a quantum)

Linux Scheduling (cont'd)



Corresponding Linux/UNIX *NICE* Values
Normal Threads Only

Some output from the shell command:

ps -eLo pid,tid,class,rtprio,ni,pri

PID	TID	CLS	RTPRIO	NI	PRI
1	1	TS	-	0	19
2	2	TS	-	0	19
3	3	TS	-	0	19
5	5	TS	-	-20	39
16	16	TS	-	0	19
17	17	FF	99	-	139
18	18	FF	99	-	139
19	19	FF	99	-	139
51	51	TS	-	0	19
52	52	TS	-	5	14
53	53	TS	-	19	0
1996	1996	TS	-	1	18
1996	2004	TS	-	0	19
1996	2005	RR	99	-	139

Linux Scheduling (cont'd)

- For historical reasons, Linux still allows the ***NICE*** value mechanism for setting timeshare priorities
  - Nice values range from **-20 to +19**
  - **-20** is HIGHEST priority, +19 is lowest
  - Most normal login processes begin with NICE == 0
  - Nice values provide an anchor point for a timesharing thread priority, but aging can change the actual NICE based priority up or down by a true 10 units
    - A thread with a NICE == 0 has a true base priority of 19
    - Aging can drive it up to 29 or down to 9

Linux Scheduling (cont'd)

- For each schedulable core, the scheduler maintains 2 sets of 140 queues
  - The **active** set is currently in use for that core
  - The **expired** set contains threads that have expired their execution quantum
    - Threads that have been **identified as interactive** may actually be given more than one quantum before being placed on the expired queue set
  - When all of the threads from the active queue set have been moved to the expired queue set, the sets are switched
    - Expired becomes active and active becomes expired
    - This gives even low priority threads a chance to run

Scheduling Affinity

- The scheduling code for each core in a system manages that core's set of queues
  - A core uses its dedicated 140 queues to select a thread to run on a context switch
  - When a new thread is created, its **core set attribute** determines which cores' active ready queue set that thread can be sent to
    - Load factor on a core will resolve ties
  - Once a thread is affined to a specific core, the operating system will always attempt to re-queue the thread to that core's active queue set whenever the thread is moved to the ready state
    - This is a performance choice to leverage cache footprint
    - Depending on load pattern, sometimes the OS poaches

Deadlock

- A computer system can be abstractly represented by a pair of sets (Σ, Π), where
  - Σ = {All possible allocation states of all system resources}
  - Π = {Threads}
- Threads behave like functions, mapping one system state to another as they execute
- We say that a thread is blocked if it is in a system state from which it cannot run
- We say that a thread is deadlocked if a thread is blocked in the current system state, and in all future states the system can ever reach

Deadlock

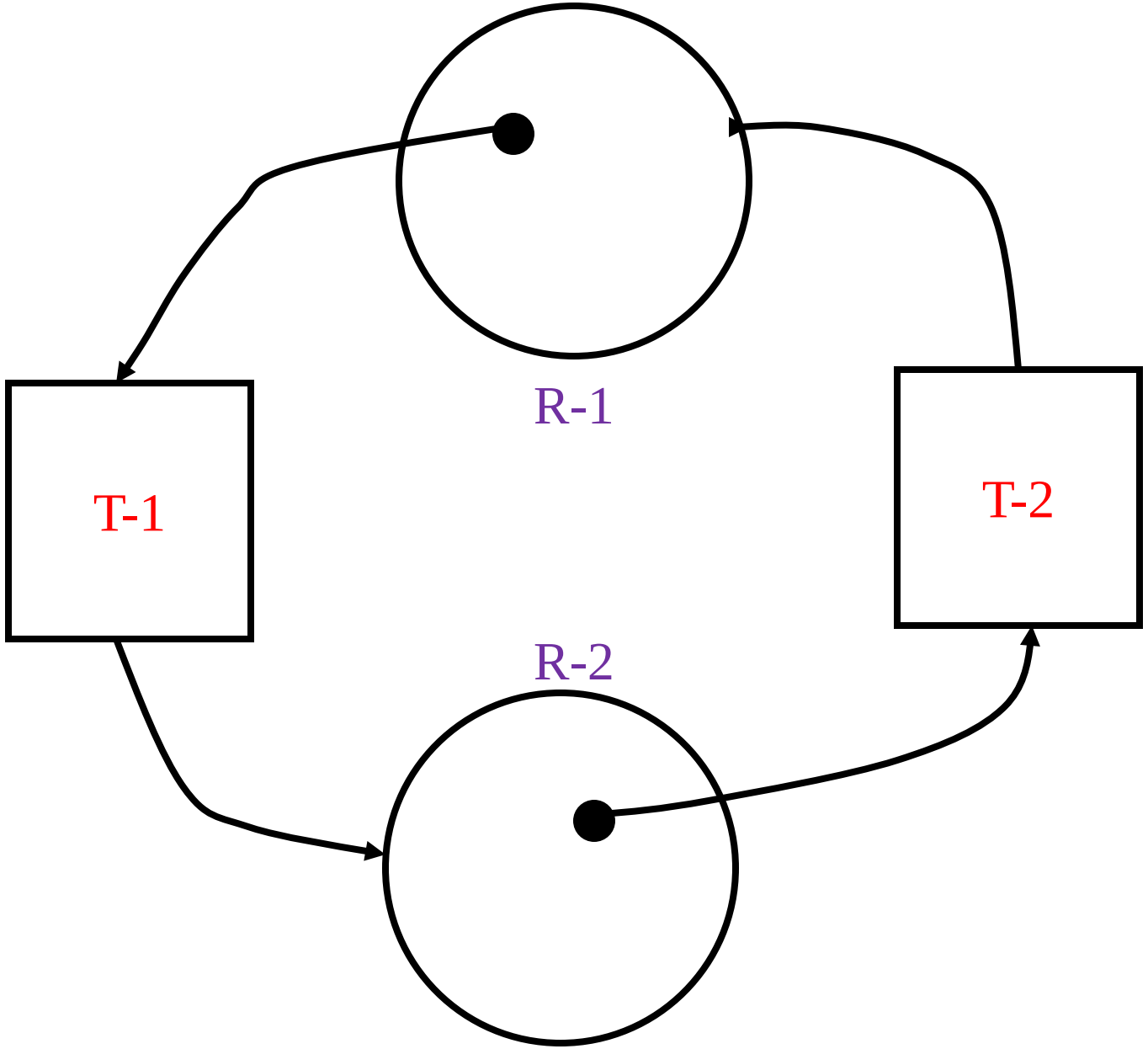
- There are 4 necessary conditions for a deadlock to occur
  - The existence of mutually exclusive resources in a system (the mutex condition)
    - Such resources are broadly characterized as either serially reusable, or consumable
  - A hold-and-wait condition in the system
  - A no-preemption condition in the system
  - A circular wait condition in the system

Deadlock

- There are 4 areas of deadlock study that have been researched extensively:
  - Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection
    - Deadlock recovery as an extension of detection

Deadlock

- Prevention involves denying a necessary condition and is always “expensive”
- Avoidance employs policy decisions which may hold-back resources to maintain “safe states”
- Detection is generally achieved by the construction and reduction of Resource Allocation Graphs (RAGs … bipartite graphs with thread and resource nodes)
- Recovery generally involves thread termination and is often based on ad-hoc policies at a given site



A Resource Allocation Graph

Deadlock

- Prevention may be achieved by denying any one of the necessary conditions:
  - Exclusively accessed resources
    - since things as basic as a memory location can fall in this category, we have to live with this condition
  - Hold and wait condition
    - a-priori resource allocation (the policy employed can lead to its own deadlock)
    - resource under-utilization (RU)

Deadlock

- Prevention (continued)
  - No preemption
    - lost work
    - indefinite postponement (IP)
  - Circular wait
    - appropriate resource ordering
    - RU
    - changes may go all the way back to application sources

Deadlock

- Avoidance
  - Safe and unsafe states
    - no single resource allocation can lead directly to deadlock from a safe state
    - consider the following system of 3 threads and 10 tape drives:

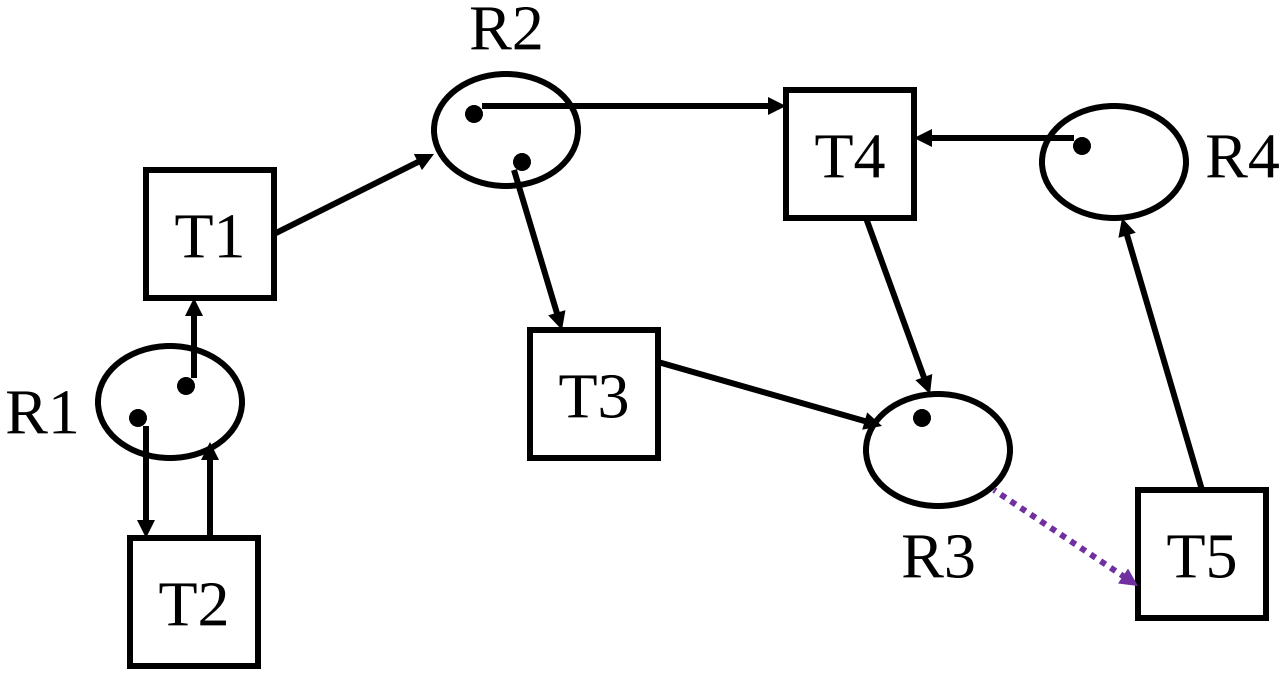
THREAD	CURRENT	MAX	BALANCE
A	2	4	2
B	3	6	3
C	3	8	5

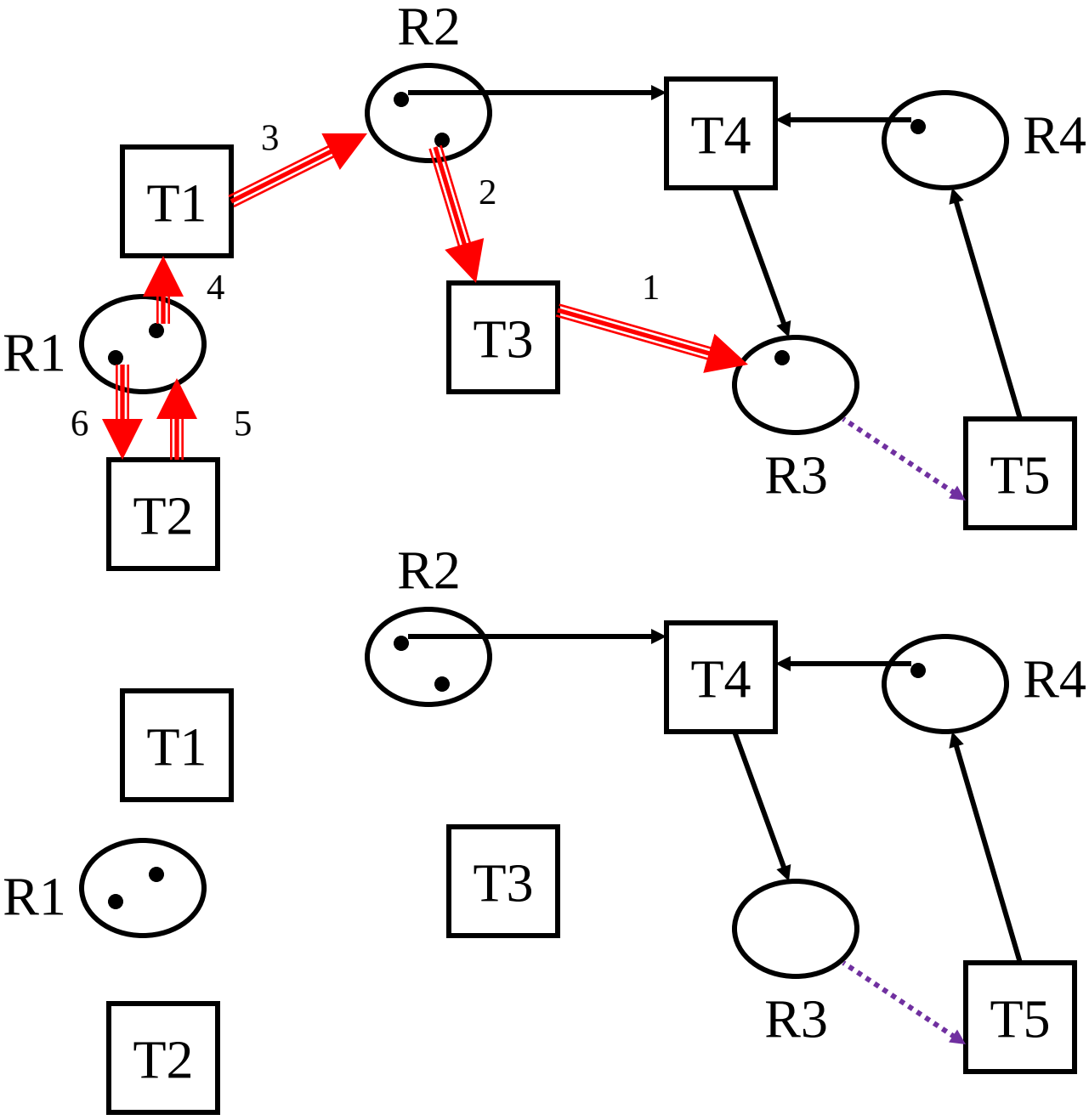
If A asks for 1 drive should the request be granted ?

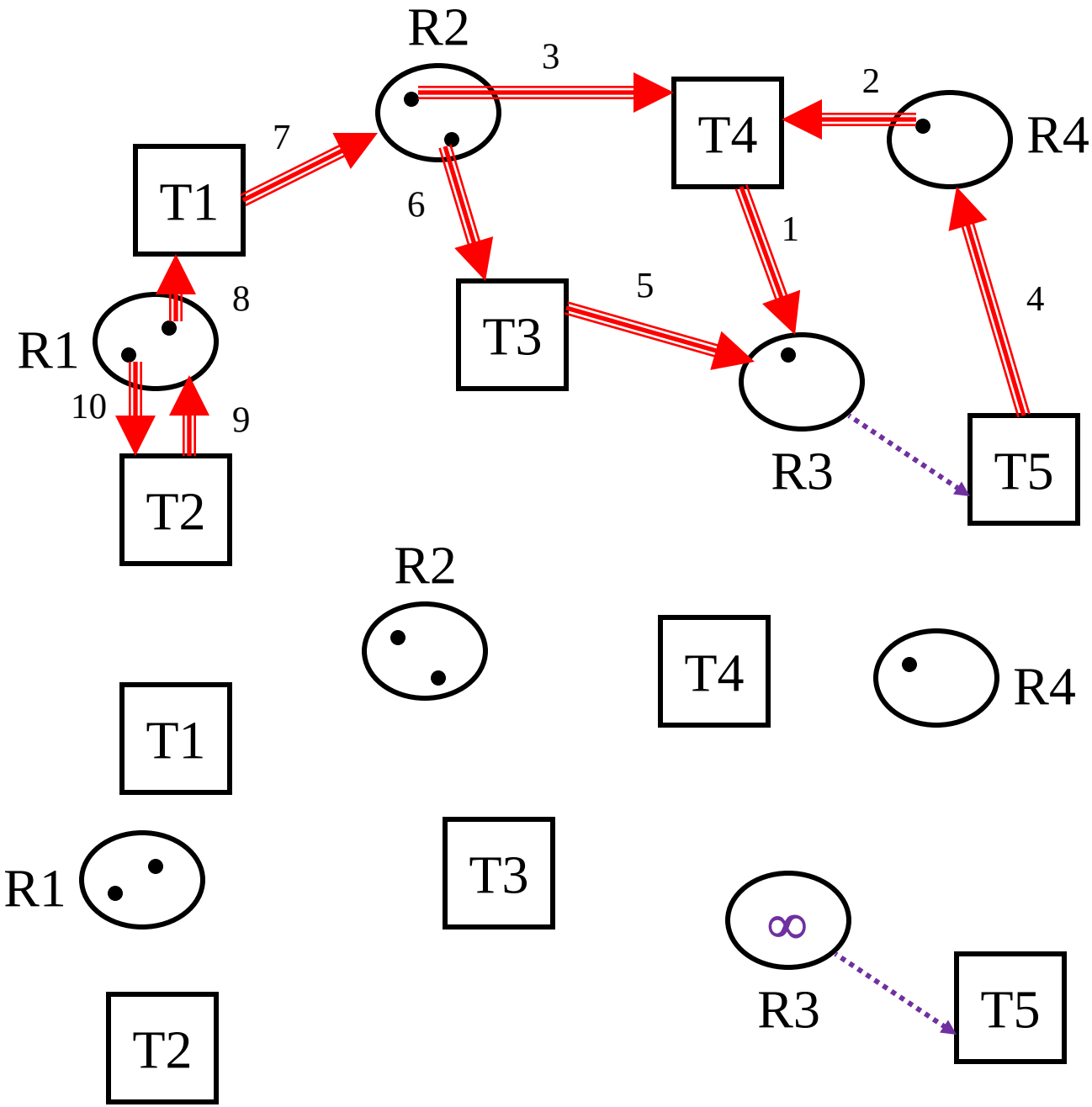
If B asks for 1 drive should the request be granted ?

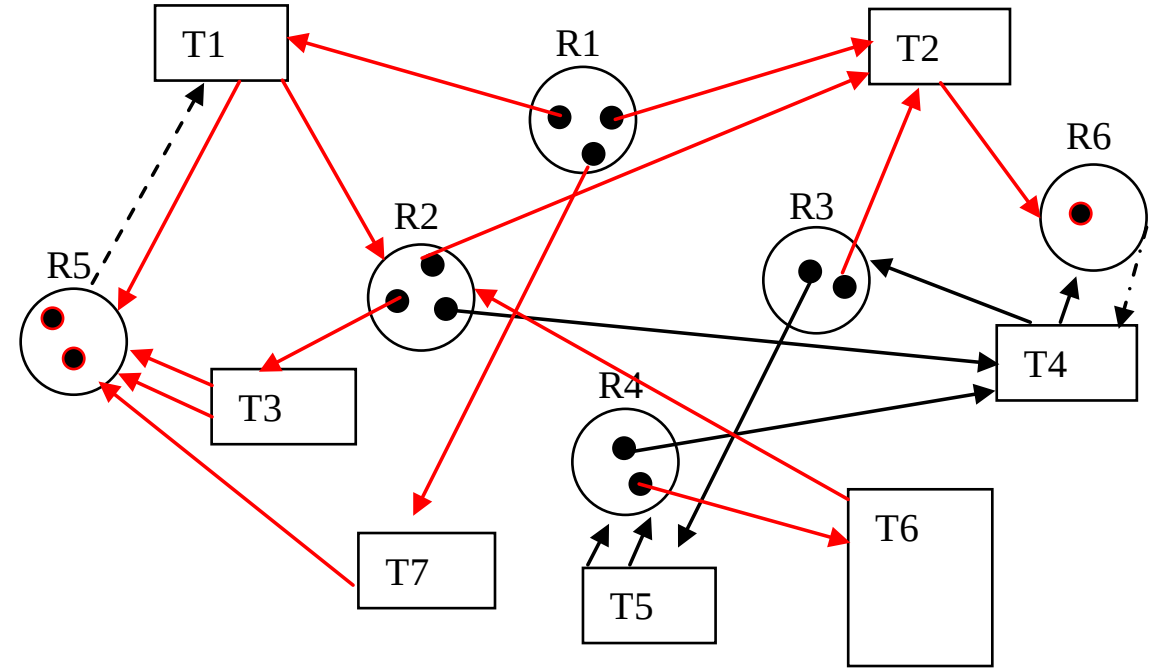
Deadlock

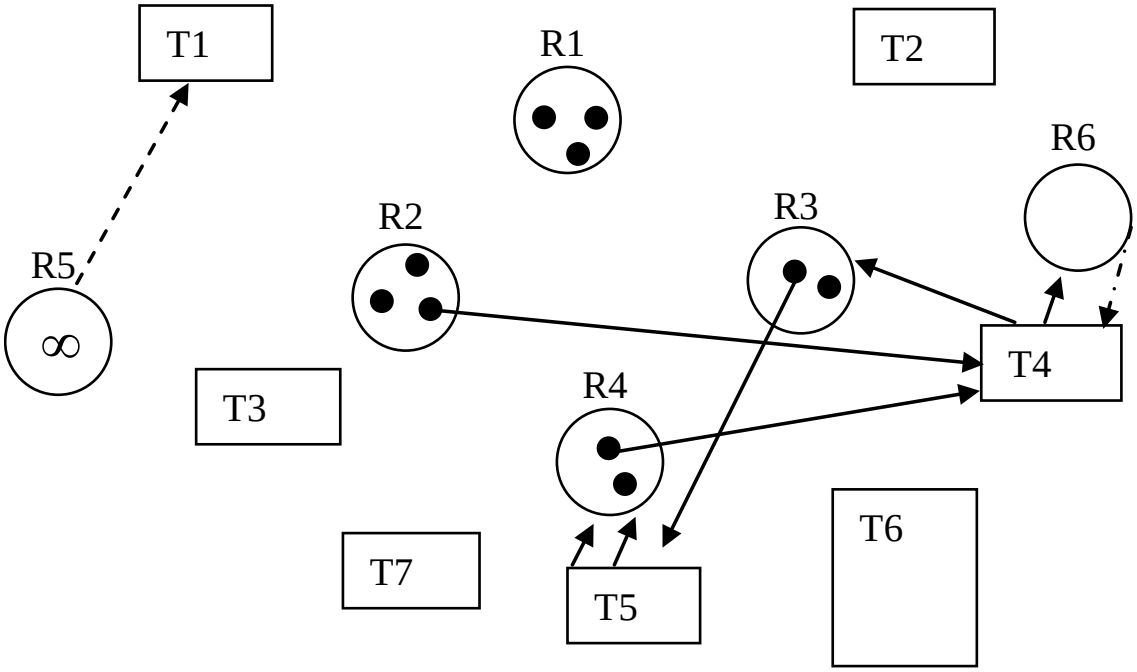
- Detection
  - RAG reduction … bipartite, M res, N threads
  - Cycle is necessary condition for deadlock in all cases, but is sufficient in **AND** model reusable only systems




R2
T4
R4
3
T1
2
4
T3
1
R1
6
5
R3
T5
T2
R2
T4
R4
T1
R1
T3
R3
T5
T2


R2
3
T4
2
R4
7
T1
6
1
T3
5
8
4
R1
10
9
R3
T5
T2
R2
T4
R4
T1
T3
R1
∞
R3
T5
T2





T4 and T5 in DL

Deadlock

- Complexity of reduction:
  - For GENERAL graphs:
    - O(MN!) (M resources and N processes)
  - For REUSABLE ONLY graphs:
    - O(MN)

Memory Management

- In describing the attributes of a process, we emphasized the address space resource
  - Each process is given a user address space container to hold its needed memory objects which minimally consist of a:
    - Text object
    - Data object
    - Stack object
- The construction and maintenance of a process address space is managed by the operating system's memory management components
- Memory management is ultimately about using the DRAM resources of a physical system efficiently

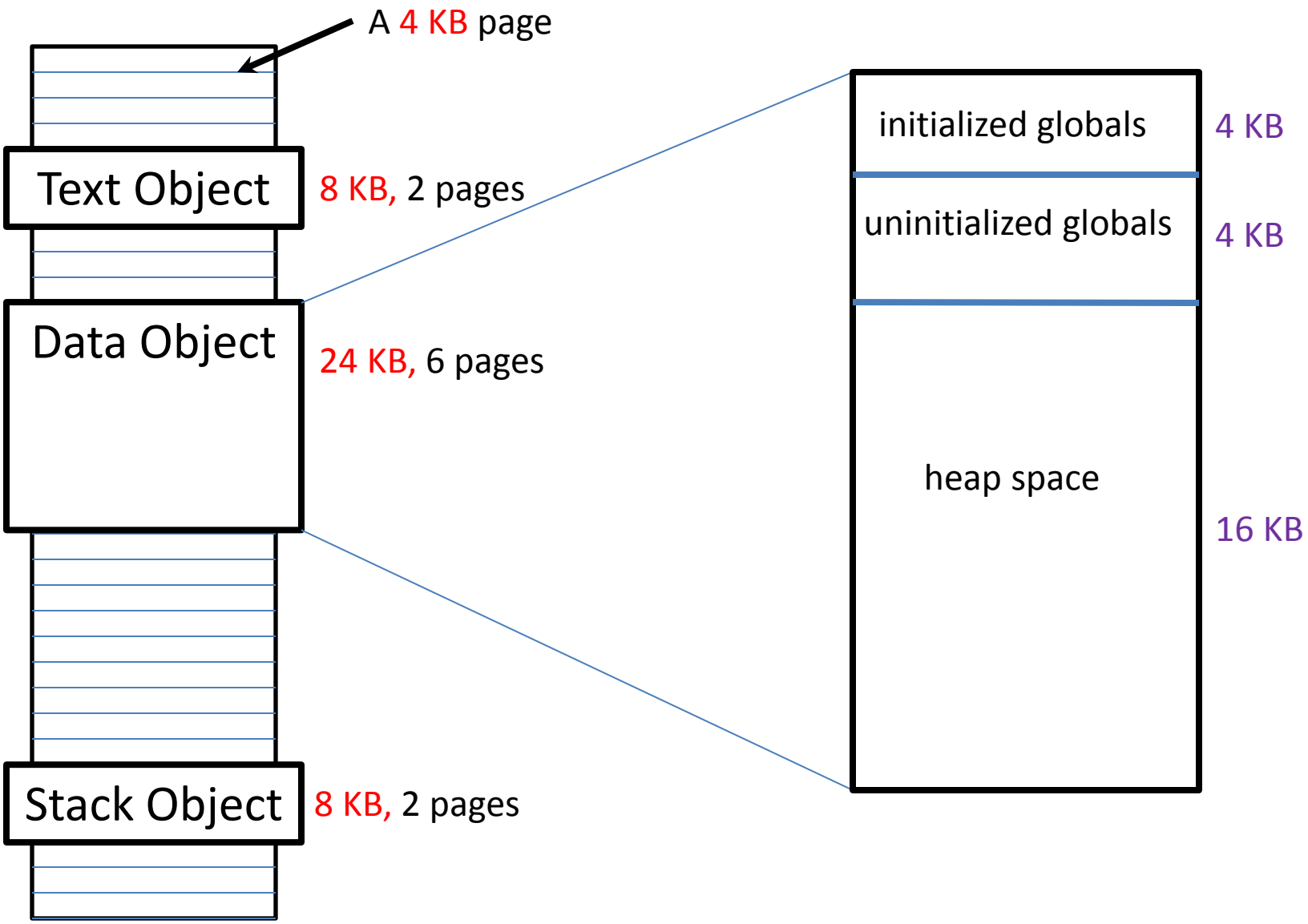
Memory Management (cont'd)

- The operating system must keep track of how each element of the physical memory is currently deployed
  - Elements may be allocated in variable sized contiguous ranges of DRAM (think malloc() here)
  - Or elements may be allocated only in multiples of 1 or more fixed sized contiguous units of DRAM commonly called page units (pages)
    - The Intel x-86 platform provides a hardware page model with a 4 KB page as its smallest page unit
    - At the physical level, if a process needs DRAM in Linux on x-86, the operating system must allocate a minimum of one page unit (4 KB)

Memory Management (cont'd)

- From a programmers perspective, allocating memory in variable sized chunks is an important abstraction
  - The libc set of memory allocation library routines supports this abstraction with calls such as ***malloc(), calloc(), realloc(),*** etc.
  - These routines depend upon the operating system to provide physical DRAM to their variable sized allocations
  - The operating system may provide 128 pages, for example, to support a 128 * 4KB = 512 KB chunk of heap, and the library routines like ***malloc()*** will manage that heap as a contiguous byte range from which variable sized allocations can be made
  - At the physical level, only 4KB pages can be allocated, but the variable sized allocations managed by ***malloc()*** happen at a higher level of abstraction layered over the physical reality

Variably sized byte regions overlaid on a fixed page size object



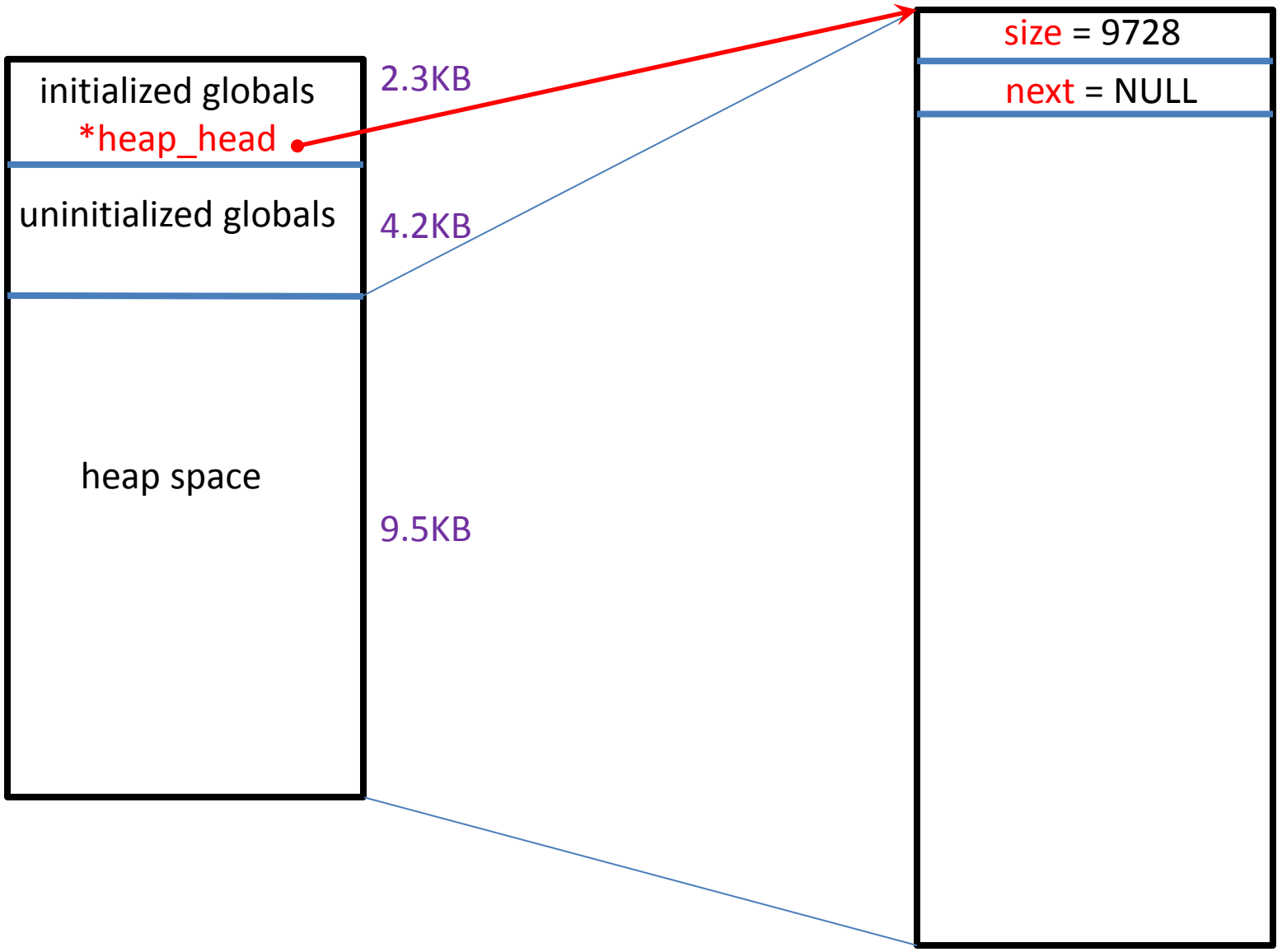
Managing Variably Sized Allocations

- Heap management library routines must keep track of the what variably sized regions are allocated and which are free in the heap
  - Various mechanisms are discussed in the book
    - Bit maps
    - Linked lists
    - Buddy system
  - While bit maps are used extensively in file system space management, they are not commonly used for address space memory management
  - We will focus on linked list and buddy system methods

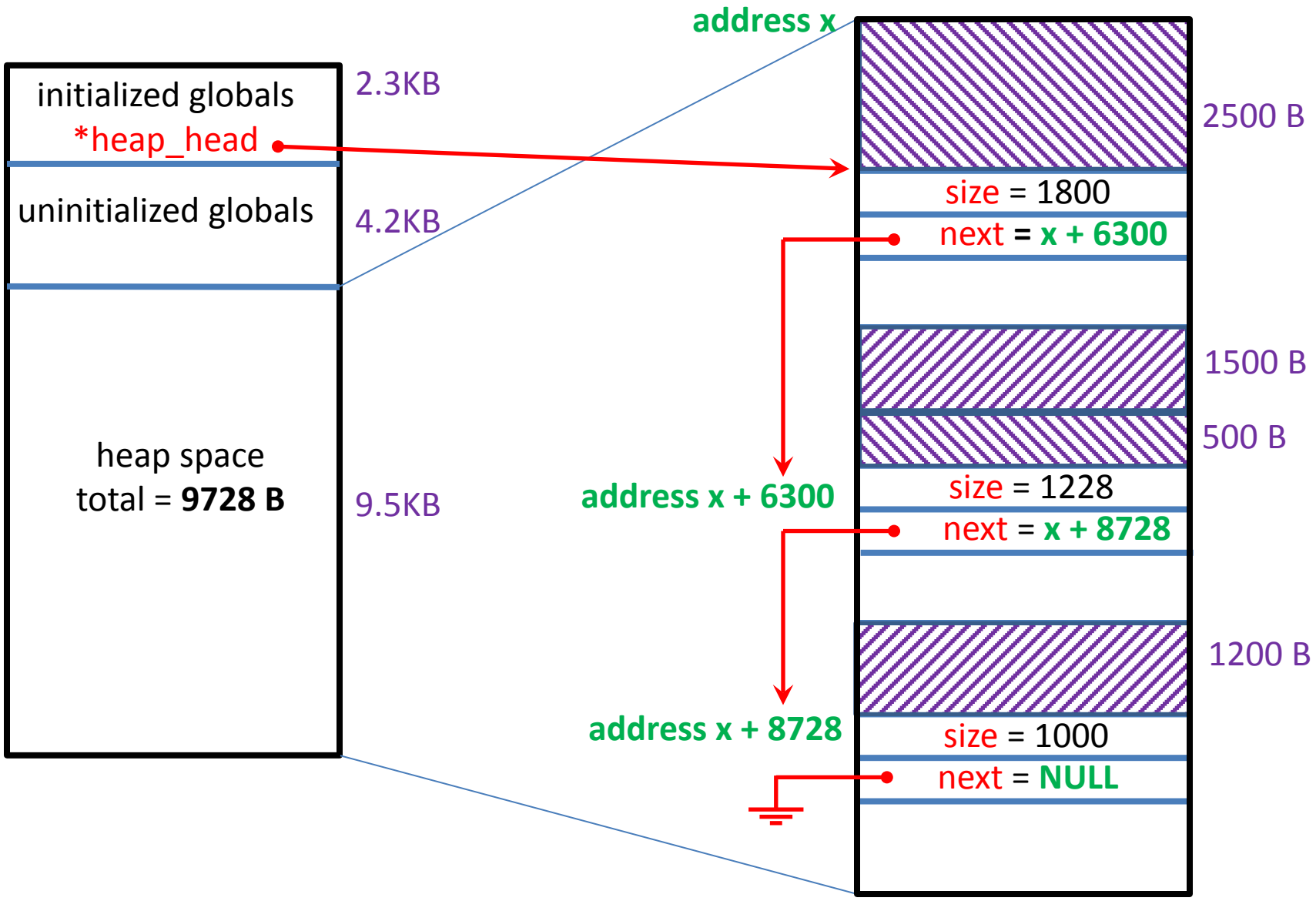
Linked List Memory Management

- The initial region is treated as a single large free element, and a named global variable points to an element header structure at the first byte location in the region
- The header structure contains a size field and a forward pointer to support a linked list
  - Initially, the size field is the size of the entire heap
  - The forward pointer is initially NULL
  - As allocations are made and freed in the region, a linked list is formed to keep track of the space

Initial configuration of the heap management list



Heap management list after allocations and frees



Linked List Memory Management (cont'd)

- The list is kept in address order as shown
- A request for an allocation of N bytes:
  - Begins with size check at *heap_head
  - If the first element is too small, follow to next
  - When an element is big enough
    - Allocate the part of the element needed (return address of this element)
    - Build a new header for the remaining area (this is the external fragment) if there is any
    - Link the fragment into the linked list

Linked List Memory Management (cont'd)

- An element must be large enough to be selected, but various algorithms use different selection criteria in an attempt to get best space utilization
  - First fit: select the first block that fits, leave any fragment on the list
    - O(n) worst case O(n/2) average case
  - Best fit: find the block that leaves the smallest (or no) fragment (small fragments give better utilization ?)
    - O(n) worst case O(n) average case
  - Worst fit: find the block that leaves the largest fragment (big fragments give better utilization ?)
    - O(n) worst case O(n) average case

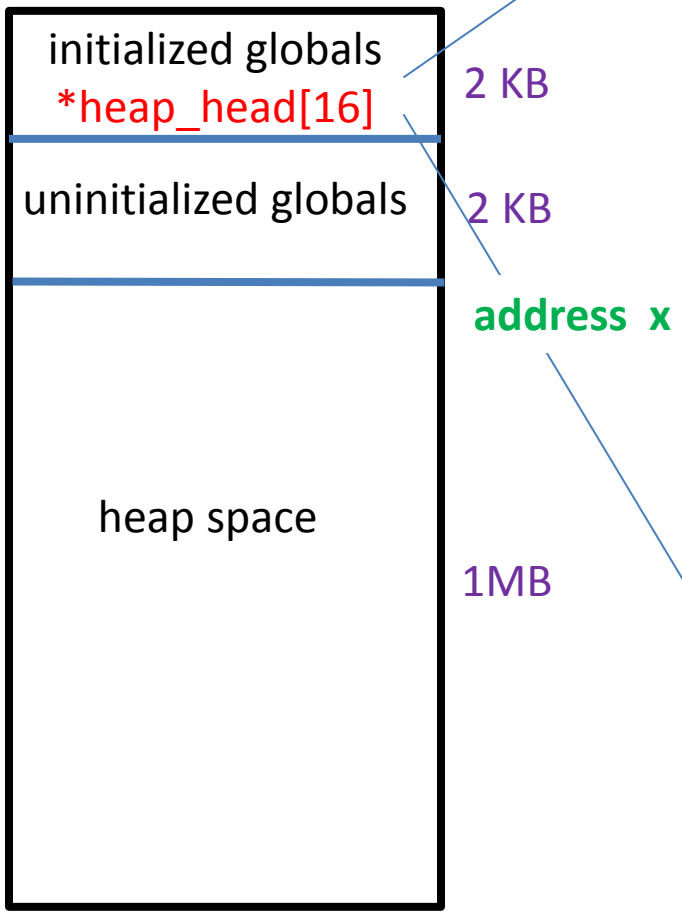
Buddy System Management

- Linked list management is simple and space efficient, but performance is O(n) worst case
- The buddy system provides O(log n) performance
  - All allocations must be a power of 2 size
  - If a request is not a power of 2 size, it must be rounded up to the nearest power of 2 size that is greater than the request size
    - A request for 3000 bytes will allocate 4096 (2^{12}) bytes
  - Oversize allocations lead to internal fragmentation

Buddy System (cont'd)

- The initial region for allocations must be a power of 2
- Keeping track of allocations is done using an array of linked lists, one for each power of 2, from the initial total size, to the smallest allowable allocation
  - A region of 1 MB (2^{20}) used for allocations, would have 16 lists assuming the smallest allocation we support is 32 Bytes
    - 2^5 = 32 Bytes, 2^6 = 64 Bytes, .. , 2^{20} = 1 MB

Initial state of lists



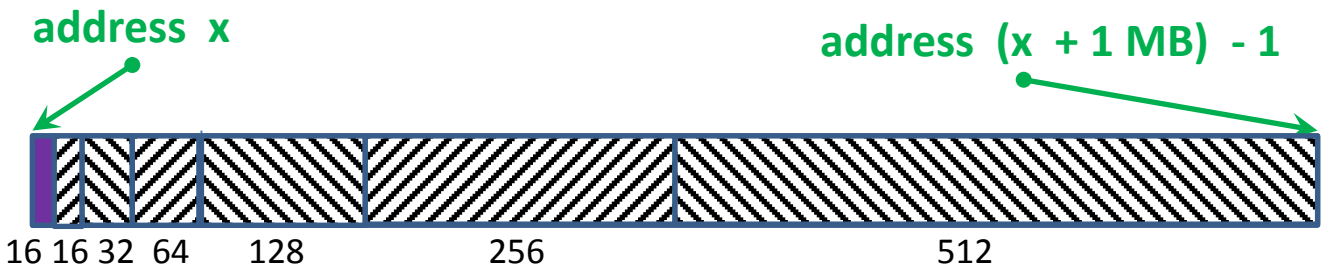
2^5	32 B
2^6	64 B
2^7	128 B
2^8	256 B
2^9	512 B
2^{10}	1 KB
2^{11}	2 KB
2^{12}	4 KB
2^{13}	8 KB
2^{14}	16 KB
2^{15}	32 KB
2^{16}	64 KB
2^{17}	128 KB
2^{18}	256 KB
2^{19}	512 KB
2^{20}	1 MB

address = x

$2^{20} / 2^{12} = 2^8 + 1 = 257$ pages * 4096 B/page = 1,052,672 total data object size

Request #1 is for a 13 KB allocation
It must be rounded to 16 KB

2^5	32 B
2^6	64 B
2^7	128 B
2^8	256 B
2^9	512 B
2^{10}	1 KB
2^{11}	2 KB
2^{12}	4 KB
2^{13}	8 KB
2^{14}	16 KB
2^{15}	32 KB
2^{16}	64 KB
2^{17}	128 KB
2^{18}	256 KB
2^{19}	512 KB
2^{20}	1 MB



1 MB memory region

req 1 at adr x

address = x + 16 KB

address = x + 32 KB

address = x + 64 KB

address = x + 128 KB

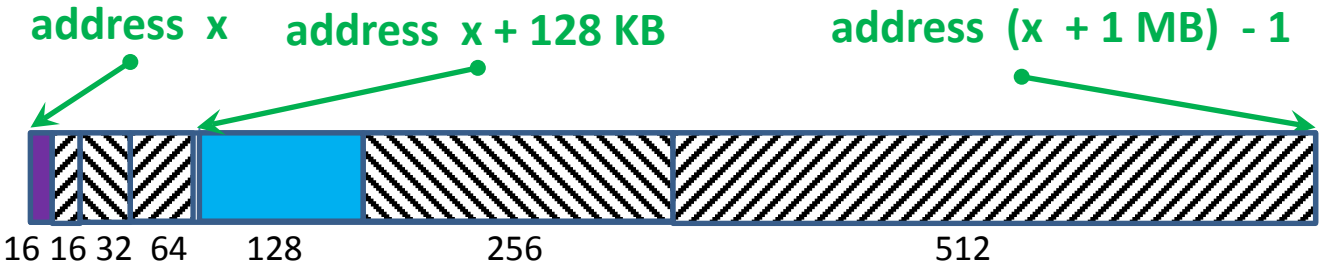
address = x + 256 KB

address = x + 512 KB

2^5	32 B
2^6	64 B
2^7	128 B
2^8	256 B
2^9	512 B
2^{10}	1 KB
2^{11}	2 KB
2^{12}	4 KB
2^{13}	8 KB
2^{14}	16 KB
2^{15}	32 KB
2^{16}	64 KB
2^{17}	128 KB
2^{18}	256 KB
2^{19}	512 KB
2^{20}	1 MB

Request #2 is for a 100 KB allocation

It must be rounded to 128 KB



1 MB memory region

req 1 at x

address = x + 16 KB

address = x + 32 KB

address = x + 64 KB

req 2 at x + 128 KB

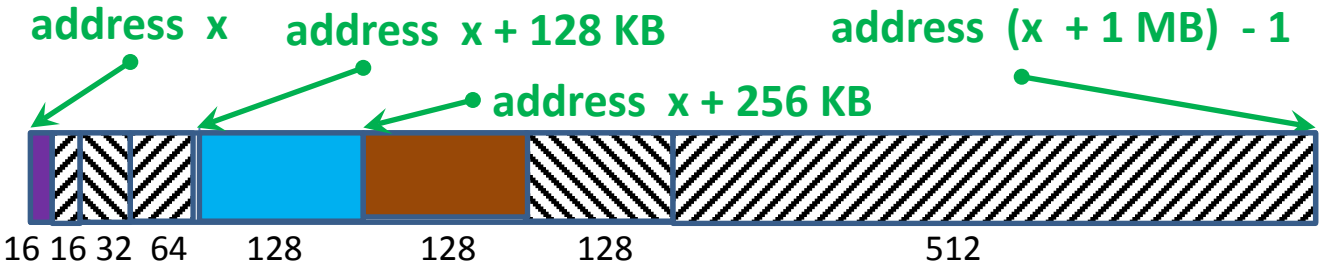
address = x + 256 KB

address = x + 512 KB

2^5	32 B
2^6	64 B
2^7	128 B
2^8	256 B
2^9	512 B
2^{10}	1 KB
2^{11}	2 KB
2^{12}	4 KB
2^{13}	8 KB
2^{14}	16 KB
2^{15}	32 KB
2^{16}	64 KB
2^{17}	128 KB
2^{18}	256 KB
2^{19}	512 KB
2^{20}	1 MB

Request #3 is for a second 100 KB allocation

It must be rounded to 128 KB



1 MB memory region

req 1 at x

address = x + 16 KB

address = x + 32 KB

address = x + 64 KB

req 2 at x + 128 KB

req 3 at x + 256 KB

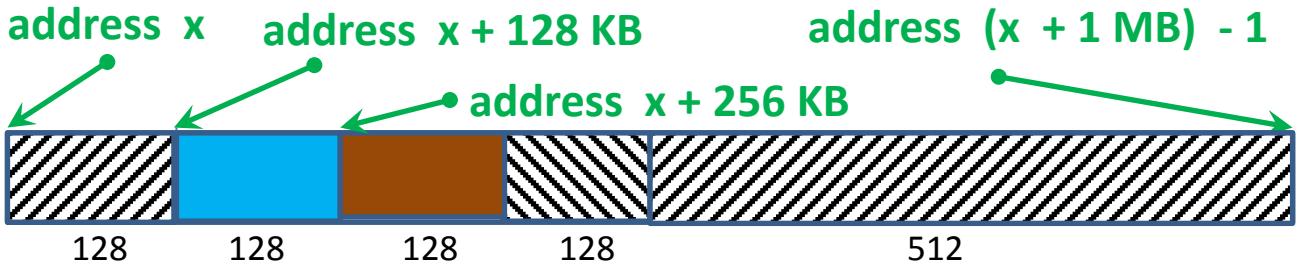
x + 384 KB

address = x + 512 KB

2^5	32 B
2^6	64 B
2^7	128 B
2^8	256 B
2^9	512 B
2^{10}	1 KB
2^{11}	2 KB
2^{12}	4 KB
2^{13}	8 KB
2^{14}	16 KB
2^{15}	32 KB
2^{16}	64 KB
2^{17}	128 KB
2^{18}	256 KB
2^{19}	512 KB
2^{20}	1 MB

Request #1 is now freed

It will buddy up with 16, 32 and 64 K



1 MB memory region

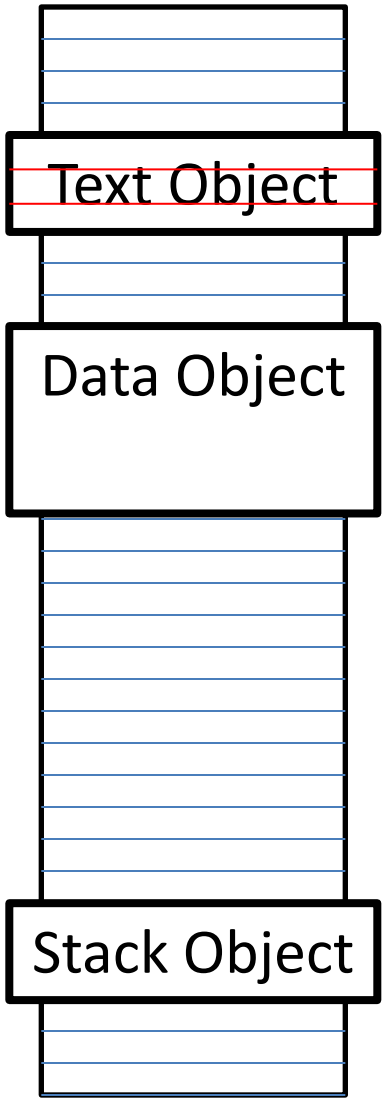
x → req 2 at x + 128 KB → req 3 at x + 256 KB → x + 384 KB

address = x + 512 KB

Buddy System (cont'd)

- The buddy system requires more meta-data than the linked list implementation
- Fragmentation here is internal and unusable
  - If a request is made for something that is not a power of 2 size, then a round-up is required
  - The extra space allocated is essentially unused but it is not available to any other request
  - **Internal fragmentation** is trading space for performance
  - when allocations are made over virtual memory, internal fragmentation is generally not a significant issue

Virtual Memory



- At the lowest memory management level, an address space is a set of virtual page frames
  - Intel x-86 systems use a 4KB page frame as the smallest supported page size
  - An address space and all associated memory objects must be multiples of 4KB
  - The number of page frames in an address space depends on the address mode
    - In **32 bit mode:** $2^{32} / 2^{12} = 2^{20} =$ **1M frames**
    - In **64 bit mode:** $2^{48} / 2^{12} = 2^{36} =$ **64G frames**
  - The OS then splits the address space into a user space/kernel space distribution

Virtual Memory (cont'd)

- The 4KB page frame size represents the minimum physical DRAM allocation that the x-86 supports when running in virtual memory mode
  - When a page of a memory object must be physically present in DRAM, the virtual page representing this region must be mapped to a physical DRAM frame that has been loaded with the required information
    - This mechanism is commonly called a page fault
  - For example, for a thread to execute code, the required code must be in a DRAM frame and that DRAM frame must be mapped to the virtual page that corresponds to the code in the process TEXT object

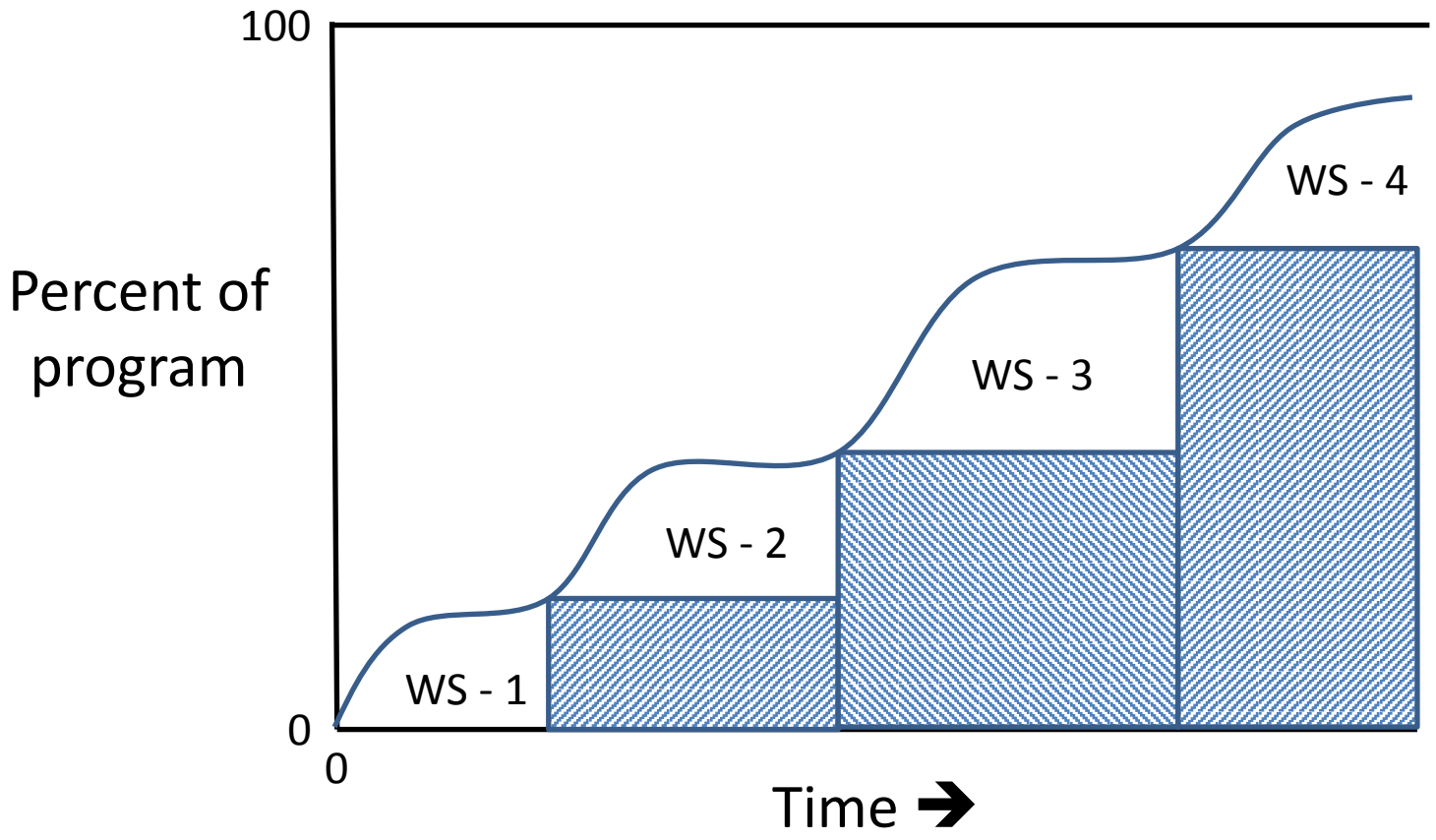
Locality of Reference

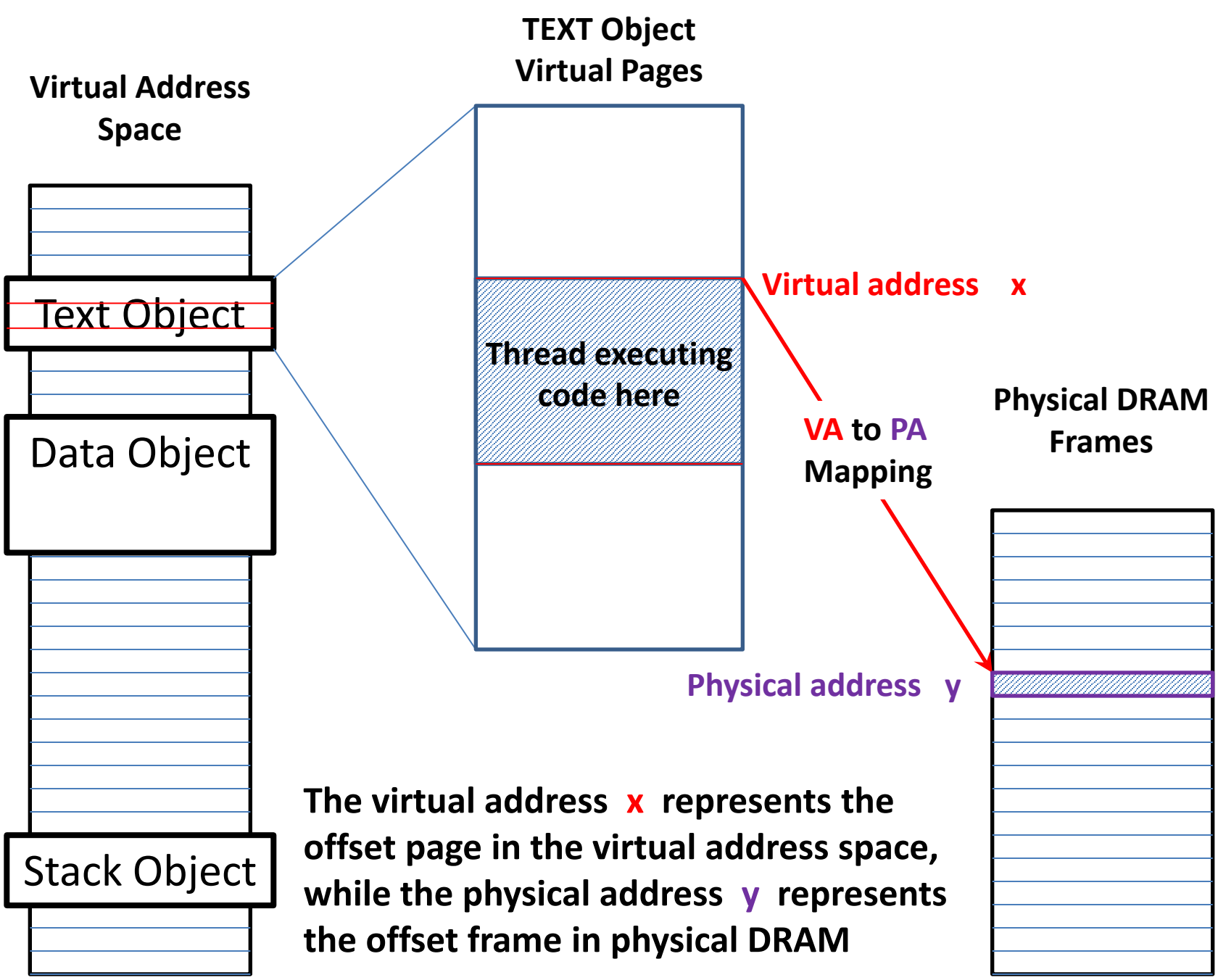
- Managing memory this way provides an efficient way to use DRAM and makes virtual memory a winning strategy
  - We use a paging policy call **Demand Paging**
    - A physical DRAM page is not mapped until some thread in the owning process makes a reference to an address on that page
    - Mapping a complete 4 KB page unit of the address space at that point makes sense, since a reference to a given address is typically followed **by more references in the same vicinity** (on the same page)
      - This phenomenon is called **Locality**
      - Programs are always organized with **spatial and temporal locality** based on how they are written and compiled

Spatial and Temporal Locality

- Programs tend to execute in a contained region at any given time
  - Such regions are called **working sets**
  - A working set consists of the code and data pages that a program is currently using
    - Spatial and temporal locality are evident in a given working set
  - As programs progress, they tend to change from one working set to another
    - Transitions are usually accompanied by a noticeable increase in the fault rate of a process
    - When the transition is complete, the fault rate tends to level off during the execution time slot of the new working set
    - Previously used pages now become likely victims for page reclamation

Working Sets

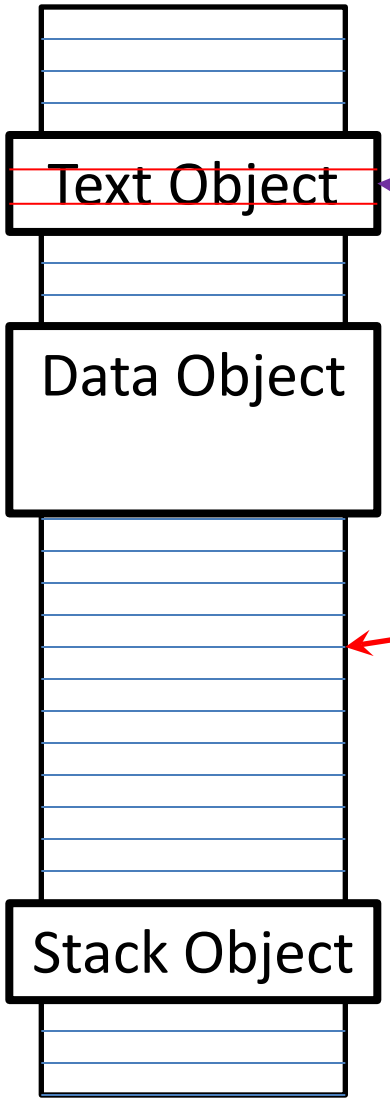



Virtual Address Space
TEXT Object Virtual Pages
Text Object
Data Object
Stack Object
Thread executing code here
Virtual address x
VA to PA Mapping
Physical DRAM Frames
Physical address y
The virtual address x represents the offset page in the virtual address space, while the physical address y represents the offset frame in physical DRAM

Virtual Memory (cont'd)

- The virtual to physical mappings for every process must be kept by the operating system
  - These mappings are kept in a per-process data structure called a page table
  - A page table has an entry for each virtual page
    - If the virtual page is currently mapped, then the entry has the physical page frame number of the mapping
    - If the virtual page is currently unmapped, then the entry has a bit to indicate "DRAM not-present" (invalid entry)
    - When a virtual page is referenced by a thread of the process it lives in, an invalid table entry causes an exception
      - If the requested page is part of an actual address space object (like the TEXT object) this becomes a page-fault
      - If the requested page is not part of any object this becomes a segmentation fault

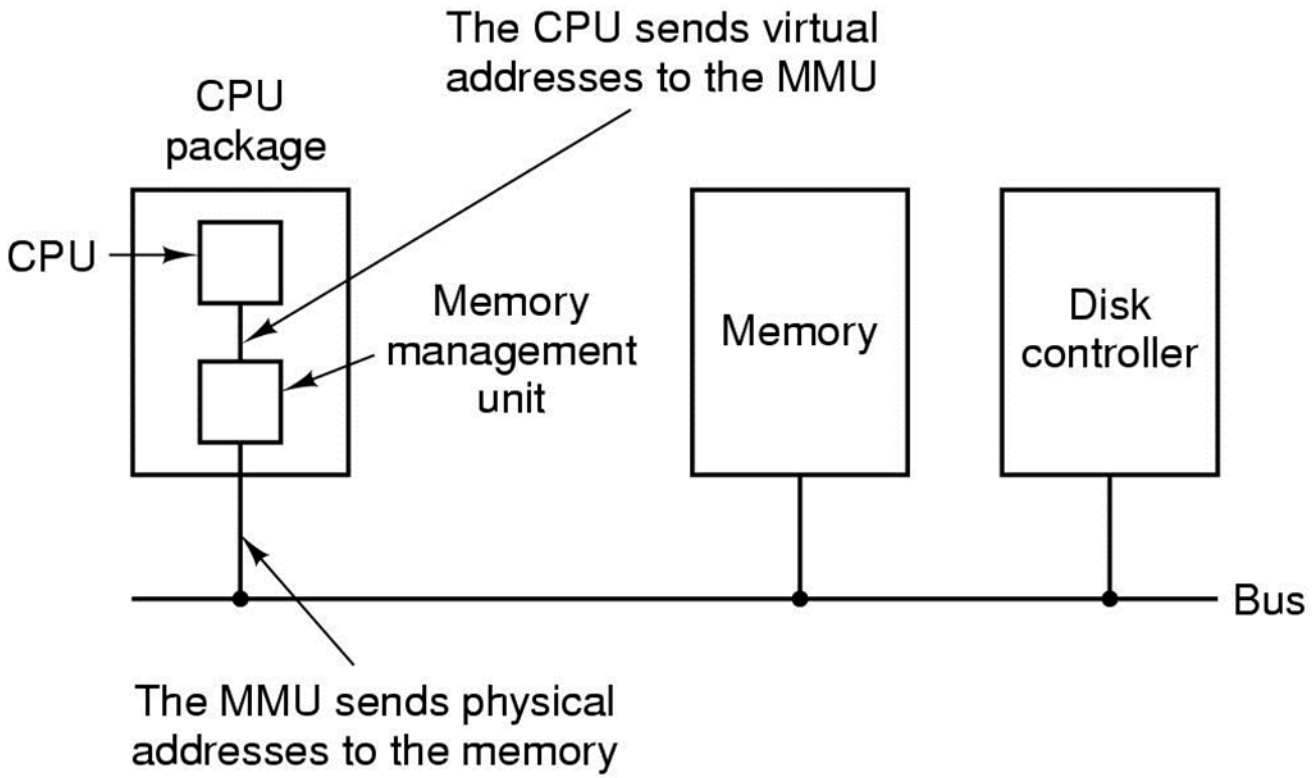
A reference to a virtual address that does not fall on a memory object is always a segmentation fault



Reference here becomes a page fault if there is no current mapping

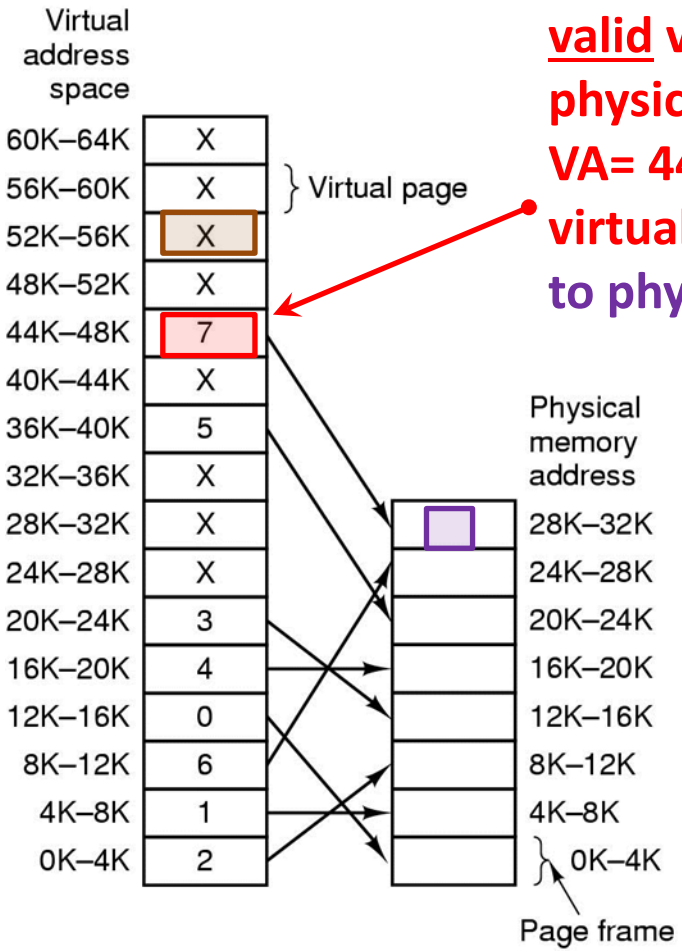
Reference here becomes a segmentation fault since there can never be a valid mapping

The Memory Management Unit



The MMU must translate all virtual address references made within the processor to physical addresses before they can be used for cache checks or external bus operations

Page Tables

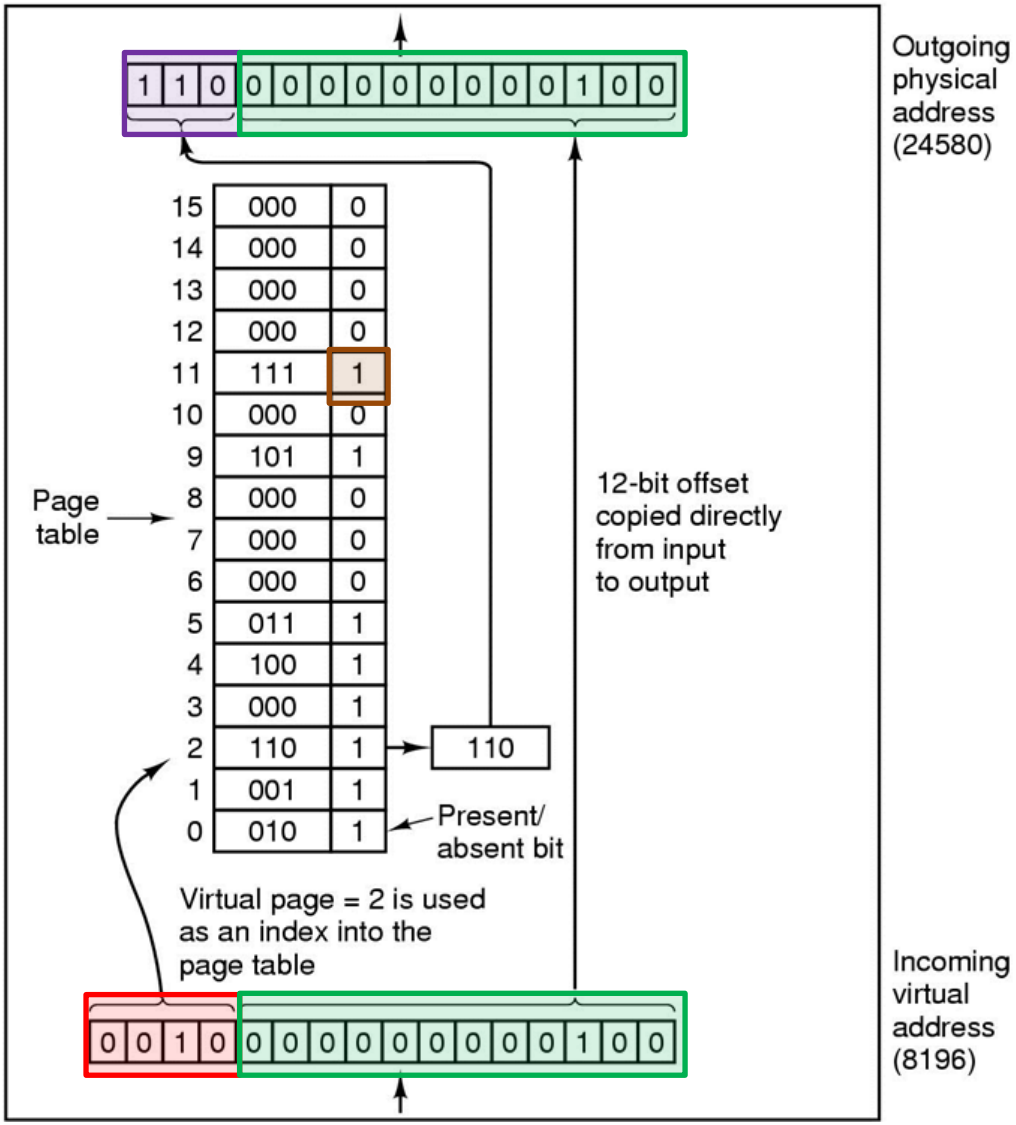


valid virtual to physical mapping
VA= 44K, PA=28K
virtual page 11 mapped to physical frame 7

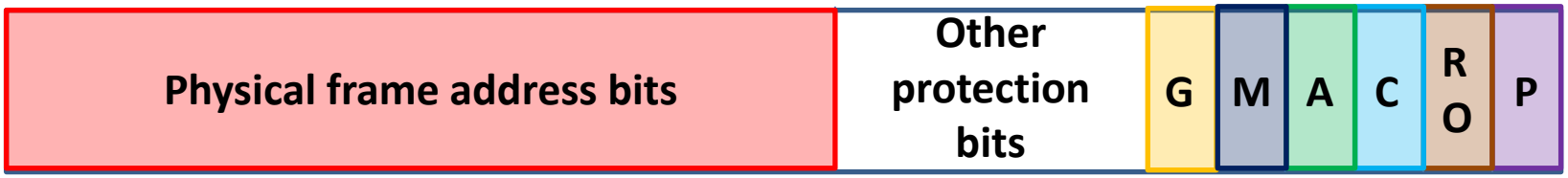
invalid virtual to physical mapping
VA= 52K unmapped
virtual page 13
not in DRAM

A Page Table is an array of Page Table Entries (PTEs) that either map a virtual page to a physical page if a mapping exits, or indicate that no current mapping exits

An example of a **16 virtual page** address space mapped into an **8 frame physical DRAM space**. PTEs have a **"present or valid"** bit to show **map state**. The 12 bit **"offset"** portion of an address points to a byte on a **4KB** page/frame



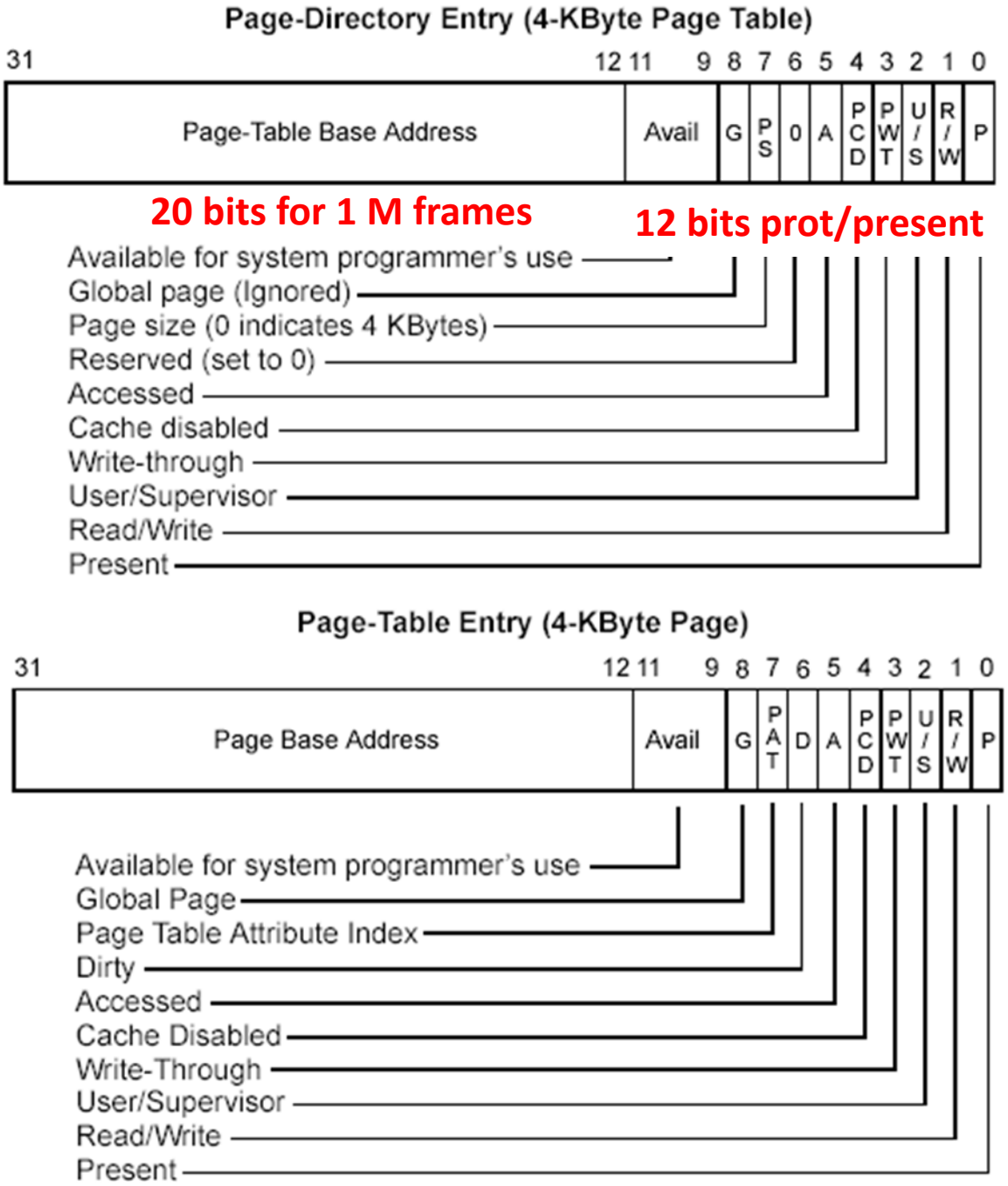
PTEs



- The PTEs that occupy a page table slot contain:
  - **Enough bits for a physical page frame address**
    - If we have 16 physical frames this would require 4 bits
    - If we have 1 million physical frames this would require 20 bits
  - **A present (valid) bit to show if a slot has a valid** translation or is currently unmapped
  - Protection bits, some of which may show
    - **If the page is read-only**
    - **If the page is cacheable**
    - **If the page has been accessed and if so, modified**
    - **If the translation is valid for all address spaces (Global)**

Page Tables (cont'd)

- Keeping a page table for each process is an expensive problem
  - In a 32 bit address space using a 4KB page frame size, it takes 1 M PTEs (as seen on slide 96)
  - If a **4 byte PTE** is required for the translation and protection/present bits, then 4 MB of DRAM would be needed for every process in the system to contain its entire page table
  - To mitigate this problem, the page table is often kept in multiple levels
    - The highest level is usually called a **directory page table**
    - The lowest level is usually called a **basic page table**
    - The directory page table must be DRAM resident at all times, and is normally the size of a single page frame (4KB for Intel x-86)
    - Basic page tables are also sized to fit one per frame
  - A 4 byte PTE allows for **1024 PTEs** per 4KB DRAM frame



Intel x-86 **PTEs** for a 2^{32} virtual address space mapped to a 2^{32} physical bus space

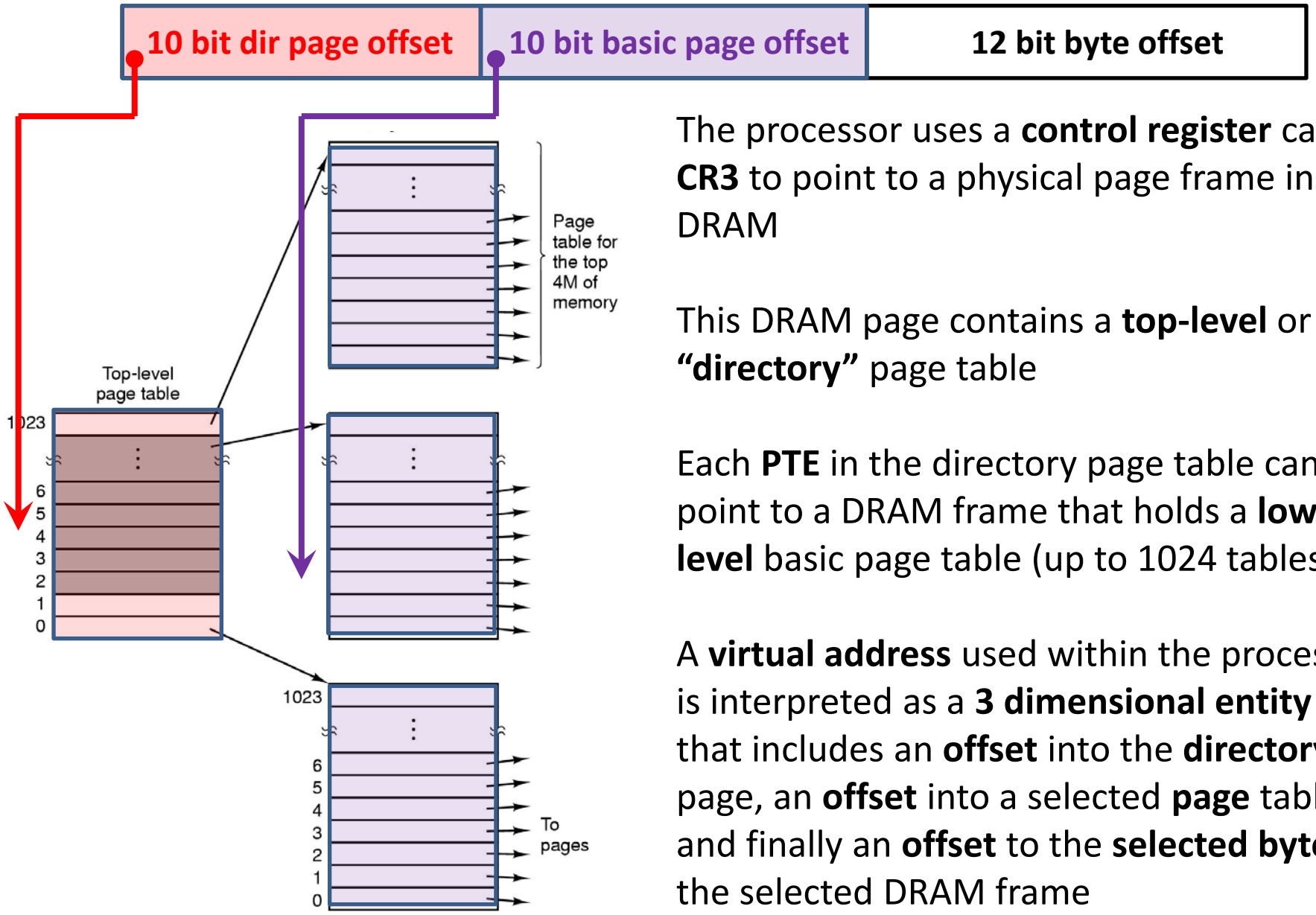
The first Pentium processors used this mapping

4 GB virtual onto 4 GB physical

PTE Control Bits

- Present flag
  - If it is set, the referred-to page (or Page Table) is contained in main memory.
- Accessed flag
  - Set each time the paging unit addresses the corresponding page frame.
- Dirty flag
  - Applies only to the Page Table entries.
- Read/Write flag
  - Contains the access right (Read/Write or Read) of the page or of the Page Table
- User/Supervisor flag
  - Contains the privilege level required to access the page or Page Table
- PCD and PWT flags
  - Controls the way the page or Page Table is handled by the hardware cache
- Page Size flag
  - Applies only to Page Directory entries.
- Global flag
  - Applies only to Page Table entries.

Intel x-86 address in 32 bit mode using a 32 bit system bus



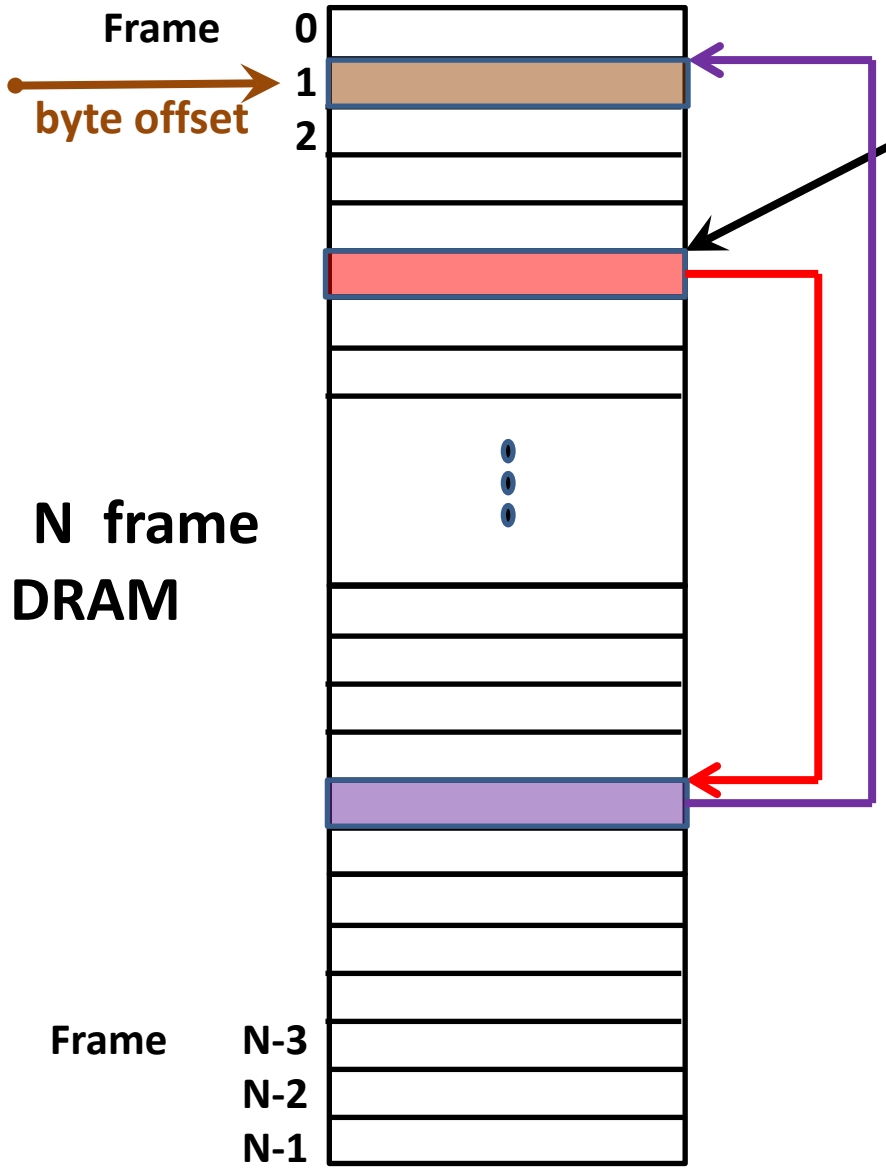
The processor uses a **control register** called **CR3** to point to a physical page frame in DRAM

This DRAM page contains a **top-level** or **"directory"** page table

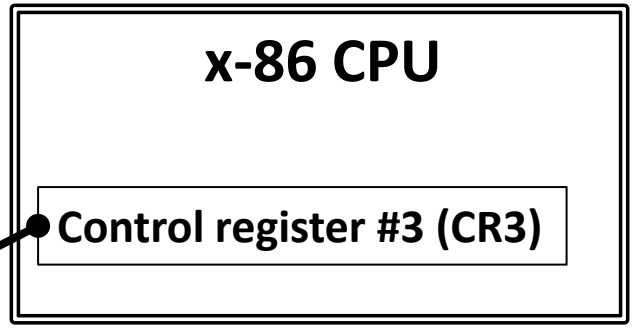
Each **PTE** in the directory page table can point to a DRAM frame that holds a **low-level** basic page table (up to 1024 tables)

A **virtual address** used within the processor is interpreted as a **3 dimensional entity** that includes an **offset** into the **directory** page, an **offset** into a selected **page** table and finally an **offset** to the **selected byte** in the selected DRAM frame

Full virtual to physical translation



An N frame DRAM



An x-86 32 bit address

CR3 locates frame with directory table

10 bits offset 1024 entries for the frame number of a basic page table

10 bits offset 1024 entries for the frame number of a user data page frame

12 bits offset the user data or instruction page frame to find the addressed byte

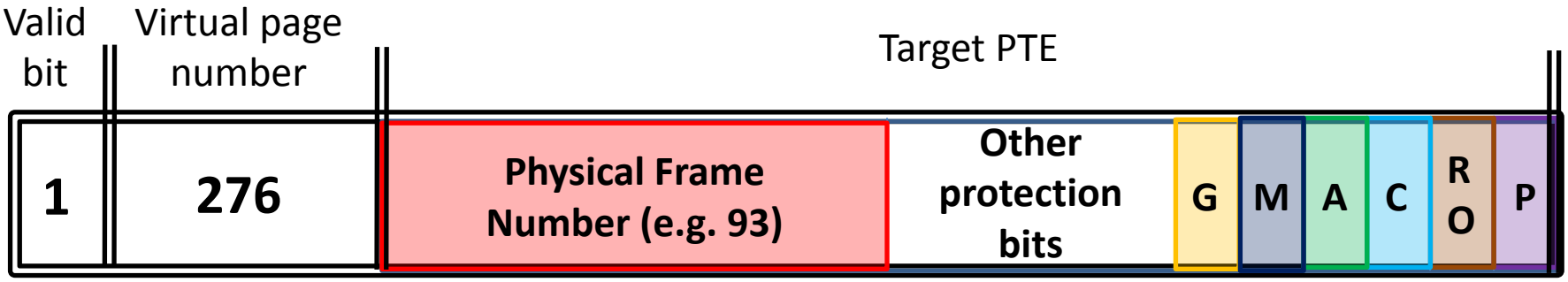
Translation Lookaside Buffer (TLB)

- The translation of a virtual address to a physical address is a tedious process
  - Various data structures must be visited
  - Any PTE that is marked invalid/not-present but is part of an object will cause a page fault to map a frame in DRAM
- Once a page table PTE provides a final mapping to a user DRAM frame of instructions/data, that information is cached into a TLB entry
  - The Translation Lookaside Buffer is a small cache
    - An entry in the cache provides a virtual page to physical frame translation
    - Because a page contains 4KBs, once mapped to satisfy a reference, we expect to need it for other close-by references **(locality)**

TLBs (cont'd)

- The mechanism for translating a virtual address to a physical address is called a **table walk**
  - A table walk begins when a reference is made to a virtual address for which there is **no current TLB entry**
    - This could be from the execution of a load/store instruction or a CPU fetch next instruction operation
    - The **TLB miss** does not necessarily mean that the target page is not in DRAM
      - Since the TLB is small, it can only hold **some of the translations** that a process may have established, but all such mapping translations must exist in the process page tables
  - The table walk **begins with CR3** and ends when the required translation is found in a PTE
    - The translation is then **placed into a TLB entry slot**
    - An **LRU eviction** policy is typically used for TLB management

A typical TLB entry for a mapping from virtual page number 276 to physical DRAM frame 93



- For an Intel TLB entry using a 32 bit mode:
  - 1 valid bit – if this bit is zero, the entry is not usable
  - 20 bits for virtual page number – the address space allows for 1 M pages
  - 32 bits for a PTE entry from the mapping page table
- The total number of slots in the TLB table varies from processor to processor, but is always small, < 1024

TLBs (cont'd)

- TLB entries are specific to a single process address space
  - A virtual page number of one process's address space may map to an entirely different physical frame than the same virtual page number in another process
  - During a Heavy Weight Context Switch, the previous process's TLB entries must all be **invalidated**
    - This is called a **TLB shootdown**
    - If the **G** bit of the PTE part of a TLB entry is on, then this entry is valid for all processes, and does not get shot down
      - The **G** bit is set on entries that map the **kernel address space**
      - All processes **share** the same kernel address space

TLBs (cont'd)

- TLB overhead is one of the most important factors in overall **system performance**
  - For reasonable CPU performance, we expect **TLB hit rates to be about 90%**
  - HW context switches destroy TLB performance, so we want to minimize these events when possible
    - Build software with multiple threads within a process rather than multiple singly threaded processes
    - **Light Weight** context switches among threads in the same process (same address space) **do not require a TLB shootdown**

Managing Physical DRAM

- Physical DRAM frames are carefully tracked by the operating system into one of three states
  - Free frames
  - Frames owned by a process
  - Frames owned by the operating system
- The operating system keeps a DRAM frame table with an entry for each physical DRAM page frame
  - This Memory Frame Table (MFT) keeps information about how each frame is currently being used
    - Which of the three states shown above is the frame in
    - For non-free frames, how often are they being touched, and have they been modified since being mapped

Memory Frame Table (MFT)

- Each entry (MFTE) in the MFT includes links so the entry can be linked onto a list
  - The Free Frame list
  - A list per PID that includes all frames that PID currently owns
  - A list of all frames the kernel currently owns
  - Various other lists and sub-lists the kernel maintains
- Whenever a page fault occurs in the system, the fault handler must use the **Free** list to find a frame to map the faulting virtual reference to
  - Mapping involves updating some PTE, removing the frame from the Free list and adding it to one of the other lists
  - The operating system must keep some minimum number of frames on the Free list to be prepared for page faults

The Free List

- The size of the Free list is controlled by a high and low watermark
  - For example, on a given system the free list low watermark may be 1% of the physical frames while the high watermark may be 3%
    - If such a system has 1 million total physical frames (this would be 1 M * 4KB/frame = ~4 GB of DRAM) then we would expect ~10,000 frames or more on the Free list
    - If the free count dropped below 10,000 then the operating system would make ready a kernel thread known as the **"Page Purger"** to remove pages from their current non-free state and add them to the Free list
      - The Purger is expected to collect enough frames to **move the Free list to the high water mark** (here ~30,000 total frames)

Page Purger

- The Page Purger thread runs an algorithm knows as a **Page Replacement Algorithm**
  - There are several such algorithms in use
  - A good algorithm must **quickly choose victims** in a way that will minimize system wide page faults
    - If I remove a page from a process and that process immediately faults it back, I made a poor choice
    - If an algorithm takes a long time to make a choice, then my system performance suffers greatly (I don't want the Purger to take too much CPU time away from the user applications that are running on the system).

Page Replacement Steps

- When the Purger runs:
  - It will visit the MFTEs of pages not already on the Free list
  - It will use some evaluation mechanism to determine if a given frame is a good victim
  - When a frame is selected, the Purger must save state on the existing frame in case we need it back
    - If a frame is mapped, then it is associated with some memory object of some address space
      - For example, we may select a frame that is currently mapped to a page in the TEXT object of some PID

Page Replacement Steps (cont'd)

- Page frame attributes:
  - A non-free page is either a part of a **File** based object such as a TEXT object, or an **Anonymous** object such as a STACK object
    - A File based object can use its file as a backing store
    - An Anonymous object must use the system Swap space as a backing store
  - A non-free page is either a part of a **Shared** object or a **Private** object
    - A shared object (TEXT object) can appear in multiple address spaces simultaneously
    - A private object (STACK) can only be mapped into exactly one address space
  - A non-free page is either **clean** or **dirty**

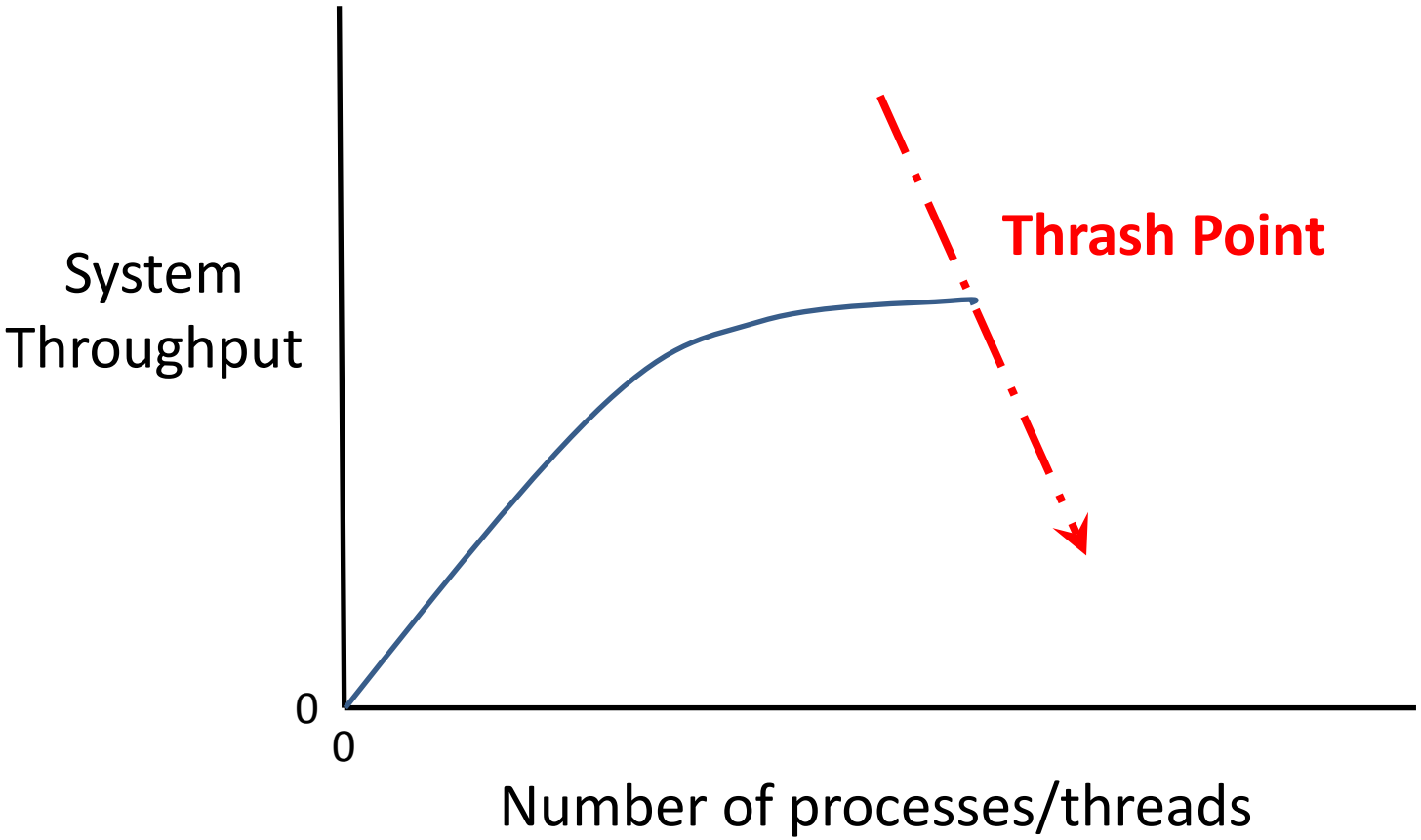
Page Replacement Steps (cont'd)

- When a Purger is selecting a victim it must consider:
  - A clean frame may mean there is no state to save
  - A dirty frame typically requires a backing store operation
    - Write to a file for a file based object, write to the swap space for an anonymous object
  - A frame from an Private object means that there is only one set of page tables to update
  - A frame from a Shared object may require updating several sets of page tables
  - A frame that has been referenced recently may be hot and expected to be referenced again
  - A frame that has not been referenced recently may have been used up and may never be referenced again

Page Replacement Steps (cont'd)

- The Purger will run through the MTFEs, marking victims that will hopefully have minimal impact on system performance
  - Clean frames from Private objects with low reference counts
- If it cannot locate enough frames to get to the high water mark on its first MFT traversal it will pass through again
  - Now choosing victims with slightly worse attributes than those found on the first pass
- It will continue until it has located enough frames to reach the high watermark
- The **frequency** with which the Purger must run is a critical system parameter
  - Too often, and the system is **thrashing**, no user work done
  - The operating system may then **intervene** and kill or swap out some entire processes in an effort to recover productivity

Load Control



Free List Maintenance

- The Free list may also grow by virtue of processes finishing and exiting
  - When a process exits, any frames associated with private objects in that process address space can immediately be added to the Free list
    - Such frames are called **Truly Worthless Pages (TWPs)**
  - Frames associated with Shared objects may be in use by other processes (part of other address space mappings) and so may not be added to the Free list
  - If we have sufficient DRAM, and if processes come and go on some regular basis, we may not need the Purger to run very frequently
    - The Free list will typically remain above the low watermark without Purger intervention

Page Replacement Algorithms

- Our text enumerates several page replacement algorithms
  - The algorithms discussed choose victims by considering all non-free pages alike and using reference information to make decisions
    - As previously discussed, a real implementation would consider many more frame attributes than just reference information
  - This simplification makes it somewhat easier to understand the basic trade-offs among these algorithms

Page Replacement Algorithms (cont'd)

- We will consider just 4 of these algorithms
  - The **non-stack algorithm** FIFO – first-in, first-out
  - The **stack algorithms**
    - OPT – optimal replacement
    - LRU – least recently used replacement
    - LFU (NFU) – least frequently used (not frequently used) replacement
- We have a strong interest in the **class of stack algorithms** because of the **inclusion property** that they all display
  - A stack algorithm guarantees that adding DRAM to a system can never result in an increase of page faults in the system when running over a benchmark

FIFO Page Replacement

- Intuitive and simple
  - When you must reclaim a DRAM page, simply choose the one that's been in service the longest
    - Hopefully this page has been used as much as it ever will be
    - If the page has been here for along time however, it may be because it is in constant use
  - FIFO is not a stack algorithm and so does not have the inclusion property
    - Belady's research has shown certain paging cases where adding DRAM to a system and running a benchmark can actually show more page faults than running the benchmark with less DRAM
      - This is known as **FIFO Anomaly or Belady's Anomaly**, and re-enforces the value of a stack algorithm for page replacement

Stack Algorithms

- Algorithms that use properties of a DRAM frame for replacement selection are known as **stack algorithms**
  - When and how often a page has been referenced are the most common properties to consider
  - While pure stack algorithms are typically too expensive to implement, variants on these algorithms are found in most operating systems
    - Recording the data to evaluate and the cost of executing the evaluation code are the main expenses

OPT Stack Algorithm

- We are interested in OPT, LRU and LFU:
  - **OPT** – this algorithm provides a lower bound on the number of page faults we can have for a given reference string across a given memory size
    - It is a **post-analysis tool**, and cannot be physically implemented (it makes decisions by looking into the future)
    - It collects the reference pattern of a benchmark program, and then uses post-analysis to determine the minimum possible faults for a given DRAM size
      - At each step, it looks into the **future** to determine the best current action
      - Its value is in establishing the **best possible result**
      - Realizable implementations can then be used on the same benchmark to see how they compare to an OPT result

LRU Stack Algorithm

- We are interested in OPT, LRU and LFU:
  - **LRU** – this algorithm selects replacement victims based on recent references to such frames
    - It is a **realizable algorithm**, and can be physically implemented with the use of access and modify bits
    - It analyzes the reference pattern of a benchmark program, and uses the Least Recently Used rubric to choose what to replace (looks into past reference patterns)
      - At each step, it looks to **past behavior** to determine the best current action with respect to victim selection
      - Victims are selected based on how near to the present time they were last referenced
      - LRU selection is based on the idea that a frame used in the recent past is likely to be needed in the near-term future

LFU Stack Algorithm

- We are interested in OPT, LRU and LFU:
  - **LFU** – this algorithm selects replacement victims based on past reference frequency to such frames
    - It is a **realizable algorithm**, and can be physically implemented with the use of access and modify bits
    - It analyzes the reference pattern of a benchmark program, and uses the Least Frequently Used rubric to choose what to replace (looks into past reference patterns)
      - At each step, it looks to **past behavior** to determine the best current action with respect to victim selection
      - Victims are selected based on how **often** they were referenced in the past time window
      - LFU selection is based on the idea that a frame used frequently in the past is likely to be needed in the future

OPT Replacement

ω	7	6	2	6	5	3	1	5	4	7	3	4	1	
1	7	6	2	6	5	3	1	5	4	7	3	4	1	
2		7	6	2	6	5	5	1	1	4	4	3	4	
3			7	7	7	7	7	7	7	1	1	1	3	
4					2	6	3	3	3	3	7	7	7	
5						2	6	6	5	5	5	5	5	
6							2	2	6	6	6	6	6	
7									2	2	2	2	2	
														Faults
c1														
c2				1	1	1	1	2	2	2	2	3	3	**10 / 2**
c3										1	1	1	2	
c4											1	1	1	**7 / 4**
c5														
c6														
c7														
∞	1	2	3	3	4	5	6	6	7	7	7	7	7	

LRU Replacement

ω	7	6	2	6	5	3	1	5	4	7	3	4	1	
1	7	6	2	6	5	3	1	5	4	7	3	4	1	
2		7	6	2	6	5	3	1	5	4	7	3	4	
3			7	7	2	6	5	3	1	5	4	7	3	
4					7	2	6	6	3	1	5	5	7	
5						7	2	2	6	3	1	1	5	
6							7	7	2	6	6	6	6	
7									7	2	2	2	2	
														Faults
c1														
c2				1	1	1	1	1	1	1	1	1	1	**12/ 2**
c3								1	1	1	1	2	2	
c4														**10 / 4**
c5											1	1	2	
c6														
c7										1	1	1	1	
∞	1	2	3	3	4	5	6	6	7	7	7	7	7	

File Systems

- File systems provide an indexing mechanism to support the saving and retrieval of information
  - Storage of some type is required
    - Persistent
      - Rotational disk (HDD)
      - Flash storage (SSD)
      - Other types of media
    - Volatile
      - RAM disk

File Systems (cont'd)

- Storage contains two kinds of information for a file object
  - Actual data (content)
  - Meta-data (indexing information)
- Both kinds of information are typically interleaved in the same storage medium
  - The storage medium is organized into some set of blocks or allocation units
  - Allocation units are typically a multiple of the underlying storage medium's minimum transfer unit (e.g. 512 byte disk sectors)
    - Some blocks are filled with file object data
    - Some blocks are filled with meta-data

File Systems (cont'd)

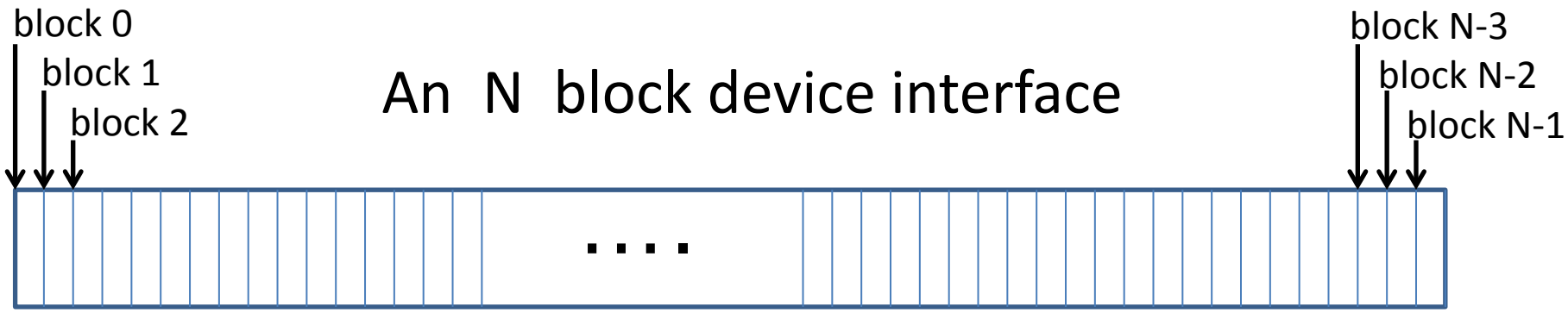
- A new file systems must be formatted onto the storage medium that will host it
  - During a formatting operation, the storage medium must be written with some initial indexing information (meta-data)
  - The data structures written during the formatting operation will eventually be responsible for indexing everything within the file system
  - The amount of information written during a format largely depends on the file system type being formatted
    - For fixed layout file systems like the Linux Ext2 and Ext3 FSs this represents a significant amount of writing
    - For dynamically evolving file systems like BTR FS or ZFS FS, the initial format is fast and minimal

File Objects

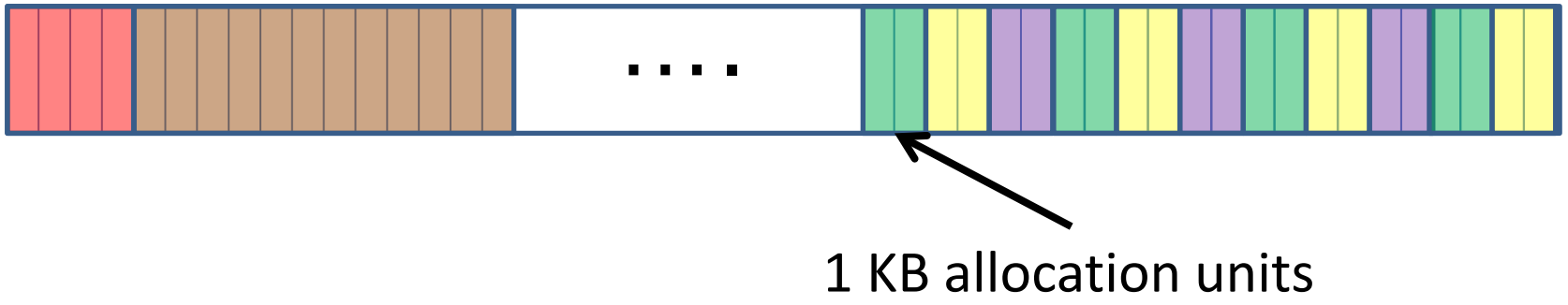
- A given file system will provide a name space for various kinds of objects
  - Ordinary text objects
  - Binary objects
  - Directory objects
  - Device objects
  - Inter-process communication (IPC) objects
  - Other possible objects
- The name space supports the location and management of these objects

Linux Ext3 File System

- To illustrate our ideas about file systems, we will look at the Linux Ext3 system in some detail
- The Ext3 file system has been used in Linux for more than 14 years
- Ext3 is a fixed format file system
  - All data structures are written to the host medium during formatting
  - Maximum number of file objects, total file system size and maximum per-object size are pre-set
  - The host medium must present a contiguous block interface (512 byte blocks) to be formatted for use
    - Typically this is either an entire HDD or a disk partition
    - The actual physical composition is unimportant, as long a block interface exists to provide the contiguous set of required blocks



- Each block is 512 bytes in size
- The blocks are grouped together during the formatting process to hold specific information
  - A collection of blocks will hold the **Super Block**
  - Another collection of blocks will hold the **I-List**
  - The remaining blocks will be aggregated into **allocation sized units** (1KB, 2KB or 4KB)



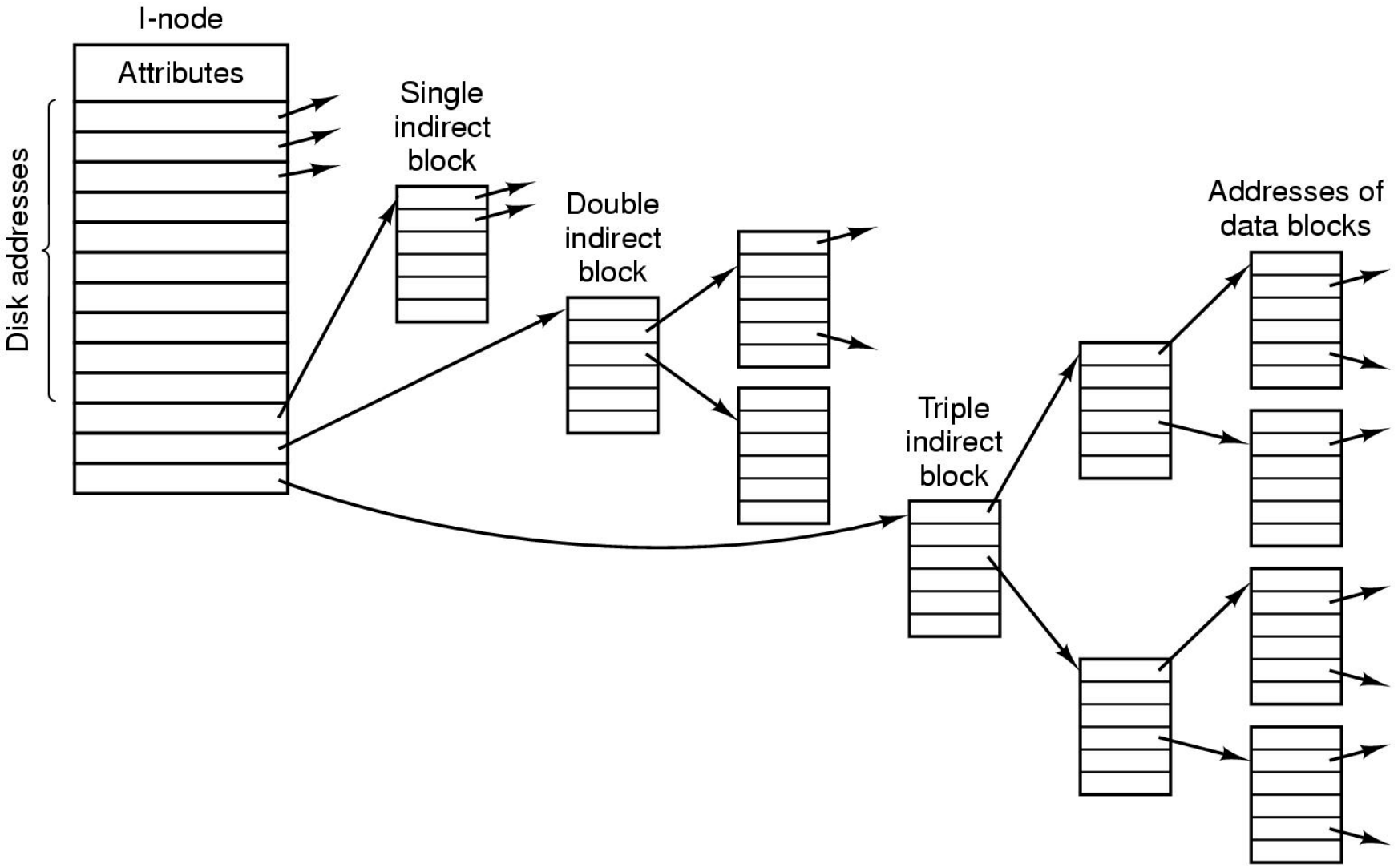
Ext3 Data Structures

- Super block
  - The super block is a table-of-contents of sorts that keeps track of the state of the file system
    - Where can we find an available I-node
    - Where can we find an available allocation unit
    - How much space is available
    - How many I-nodes are available
  - The super block is copied into kernel memory when the file system is mounted
- I-List
  - The I-List is a collection of i-nodes
    - Every file object in the file system (ordinary files, directories, etc.) is managed by exactly one i-node
    - Everything known about a file object (except a possible human readable name) is found on the object's i-node

i-nodes

- An i-node is a small data structure (typically 256 bytes) that maintains all state information for a file object
  - State include attributes such as
    - Who owns the object
    - What protection does the object have (e.g. RWX)
    - What size is the object
  - State also includes location information that specifies where the content of the file object is
    - Direct pointers that point to an allocation unit that contains some of the object's content
    - Indirect pointers that point to an allocation unit that is filled with additional pointers that point to allocation units that contain some of the object's content

i-node Structure

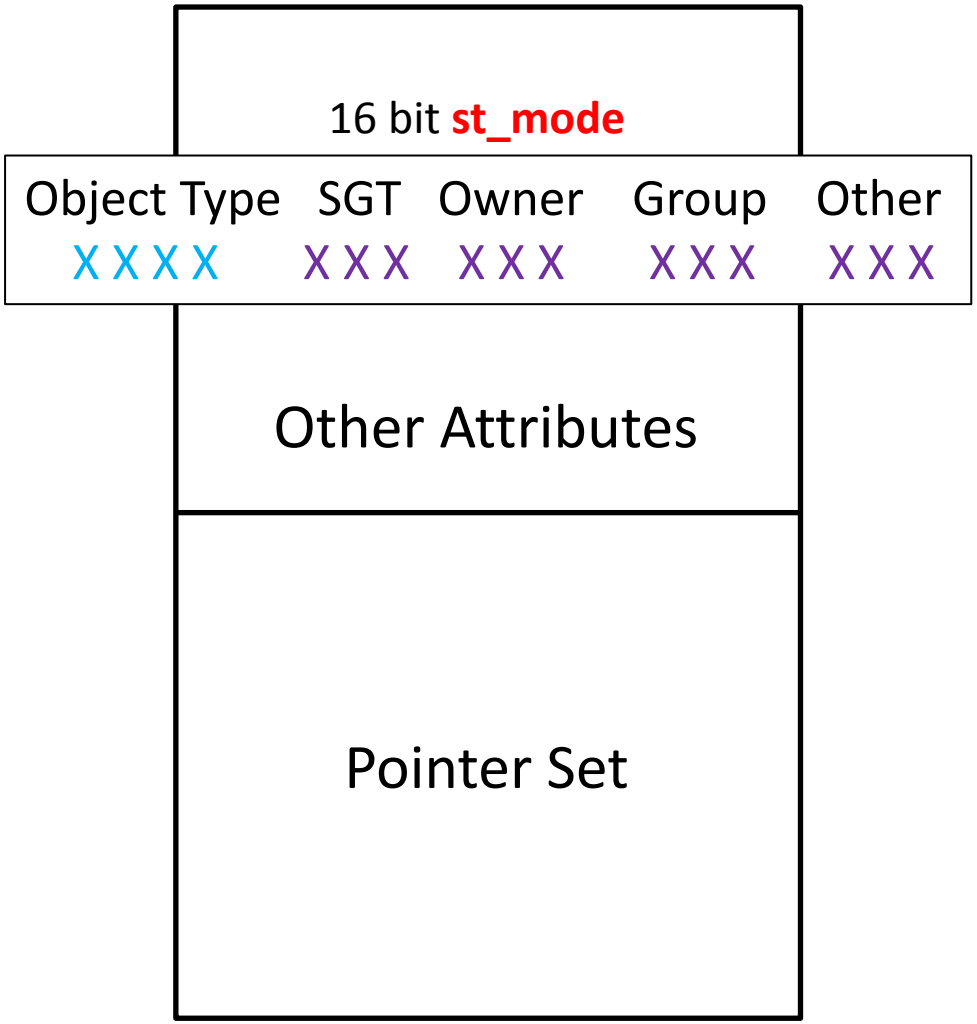


Ext3 i-node Attribute Layout

```
struct stat {
        dev_t     st_dev;       /* device inode resides on          */
        ino_t     st_ino;       /* this inode's number              */
        mode_t    st_mode;      /* protection                       */
        nlink_t   st_nlink;     /* number or hard links to the file */
        uid_t     st_uid;       /* user-id of owner                 */
        gid_t     st_gid;       /* group-id of owner                */
        dev_t     st_rdev;      /* dev type, for inode that is dev  */
        off_t     st_size;      /* total size of file               */
        time_t    st_atime;     /* file last access time            */
        time_t    st_mtime;     /* file last modify time            */
        time_t    st_ctime;     /* file last status change time     */
        uint_t    st_blksize;   /* opt blocksize for file sys i/o ops */
        int       st_blocks;    /* actual number of blocks allocated */
       ……. etc…….
};
```

```
int x;
struct stat buf;
if((x = lstat("./myfile", &buf)) == -1){
      perror("stat failed "); // etc…
```

i-node mode – protection details



Output from an ls -l Command

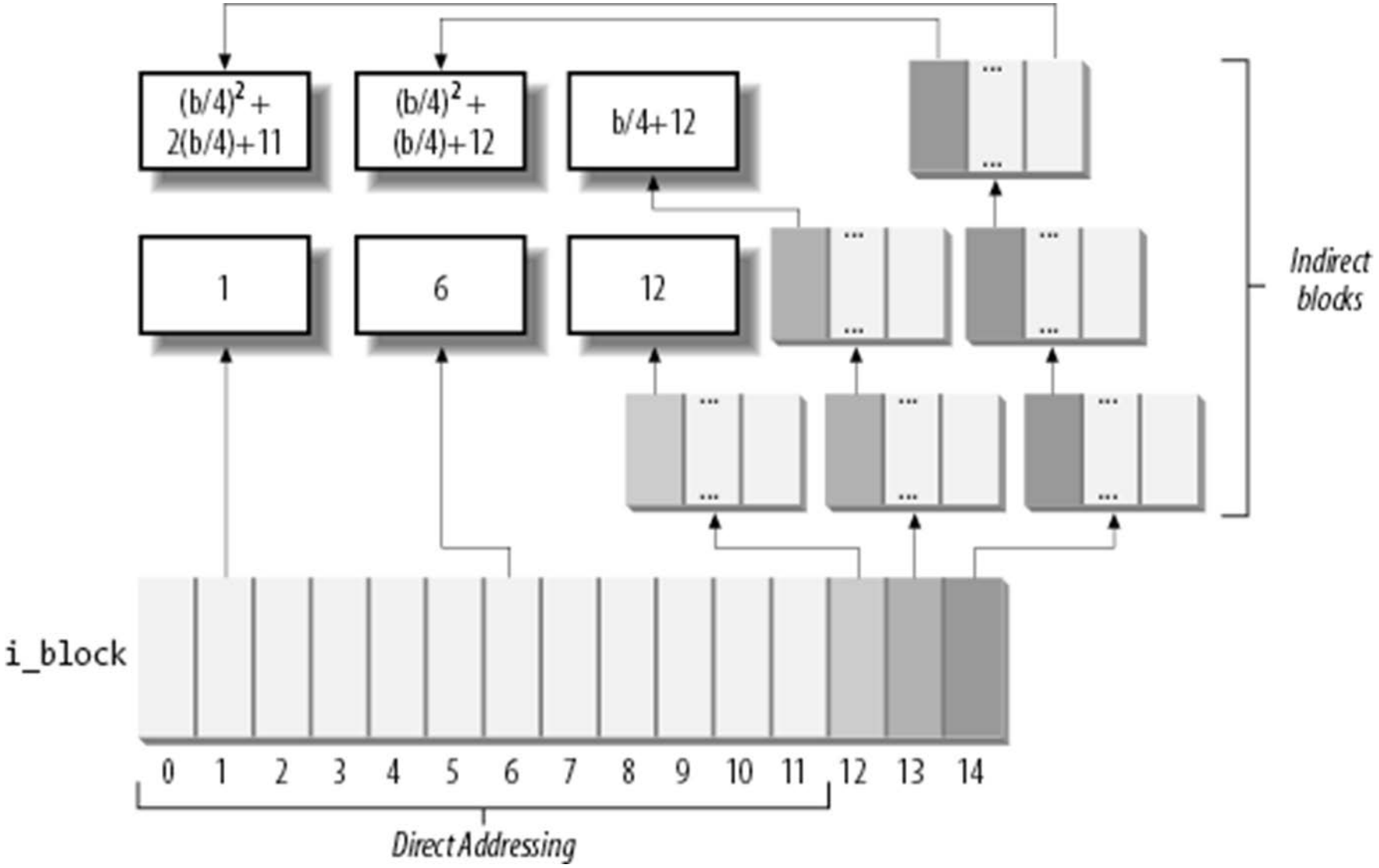
```
d rwx --- ---  2 bill   fac    4096  Jun 30  2013 dir1
- rwx --- ---  1 bill   fac    6137  Apr  5  2014 mylink
l rwx rwx rwx  1 bill   fac       4  Apr  2  2014 myls -> stat
p rw- --- ---  1 bill   fac       0  Apr  2  2014 mypipe
l rwx rwx rwx  1 bill   fac       4  Sep 24  2013 myslink -> ccnt
s rwx --- ---  1 bill   fac       0  Apr  2  2014 mysock
c rw- --- ---  1 root   root    4,3 Sep 12 11:09 /dev/tty3
b rw- rw- ---  1 root   disk    8,1 Sep 12 11:08 /dev/sda1
|  |   |   |   |  |      |       |       |           |
|  |   |   |   |  |      |       |       |           +- OBJECT NAME
|  |   |   |   |  |      |       |       +- LAST MODIFY DATE
|  |   |   |   |  |      |       +- FILE SIZE IN BYTES / MAJ, MIN #
|  |   |   |   |  |      +- GROUP AFFILIATION
|  |   |   |   |  +- OWNER NAME
|  |   |   |   +- NUMBER OF LINKS (other names to this object)
|  |   |   +- ACCESS ALLOWED TO NON OWNER NON GROUP PROCESSES
|  |   +- ACCESS ALLOWED TO NON OWNER BUT GROUP MEMBER PROCESSES
|  +- ACCESS ALLOWED TO OWNER
+- FILE TYPE AS SHOWN:        -  ordinary
                              d  directory
                              l  symbolic link
                              p  pipe
                              s  socket (UNIX domain)
                              c  character special (device)
                              b  block special (file system device)
```

```
-bash-4.1$      ls -lai

total 62
11115973941744249271 drwx------  4 bill fac  4096 Jul 25 15:49 .
 929621026346775890 6 drwx------ 18 bill fac  4096 Nov 13 21:41 ..

12289092729807287105 -rw-------  1    160 Apr  5  2010 charcnt.c
13652633959297112352 drwx------  2   4096 Jun 30  2013 dir1
13356045832246023086 prw-------  1      0 Apr  2  2010 mypipe
10357499618188756599 srwx------  1      0 Apr  2  2010 mysock
12085445459396525689 drwx------  5   4096 Apr 30  2013 mysubdir
10459140345039328129 -rwx------  2  10713 Apr 30  2013 stat
10459140345039328129 -rwx------  2  10713 Apr 30  2013 st_HL
11394210484925381891 lrwxrwxrwx  1      4 Apr  2  2010 st_SL-> stat
10629459747026869551 -rw-------  1   3800 Apr 30  2013 stat.c
12163429503322791895 -rw-------  2      0 Jan 23  2014 xyz
12009214341029995684 lrwxrwxrwx  1      3 Jan 23  2014 xyzl -> xyz
12163429503322791895 -rw-------  2      0 Jan 23  2014 xyzz
```

Ext3 i-node Pointer Layout



Ext3 File Size Limits

- The upper limit on file size depends on the allocation unit size, and the size of a pointer
  - Consider allocation unit of 4KB (1KB, 2KB and 4KB are available for Ext3 file systems)
  - Consider pointer size of 4 bytes (can point to 2^{32} or ~4 billion disk sectors)
    - 12 direct pointers * 4KB/AllocUnit = 48KB
    - 1 indirect pointer yields 1024 direct pointers
      - 1024 * 4KB = 4MB
    - 1 second level indirect pointers yield 1024^2 direct pointers
      - 1024 * 1024 * 4KB = 4GB
    - 1 third level indirect pointers yield 1024^3 direct pointers
      - 1024 * 1024 * 1024 * 4KB = **2TB (2^{41})**
        NOT 4TB (2^{42}). Half of the pointers cannot be used.
        » Note: at 512 bytes/sector we would need $2^9 * 2^{33}$ to get 2^{42}

File-size upper limits for data block addressing

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~2 TB see note

The Ext3 Journaling Filesystem

- The idea behind Ext3 journaling is to perform each high-level change to the filesystem in two steps
  - First, a copy of the blocks to be written is stored in the journal
  - Then, when the I/O data transfer to the journal is completed (in short, data is ***committed to the journal***), the blocks are written in the filesystem
  - When the I/O data transfer to the filesystem terminates (data is ***committed to the filesystem***), the copies of the blocks in the journal are discarded

Recovery From System Failure

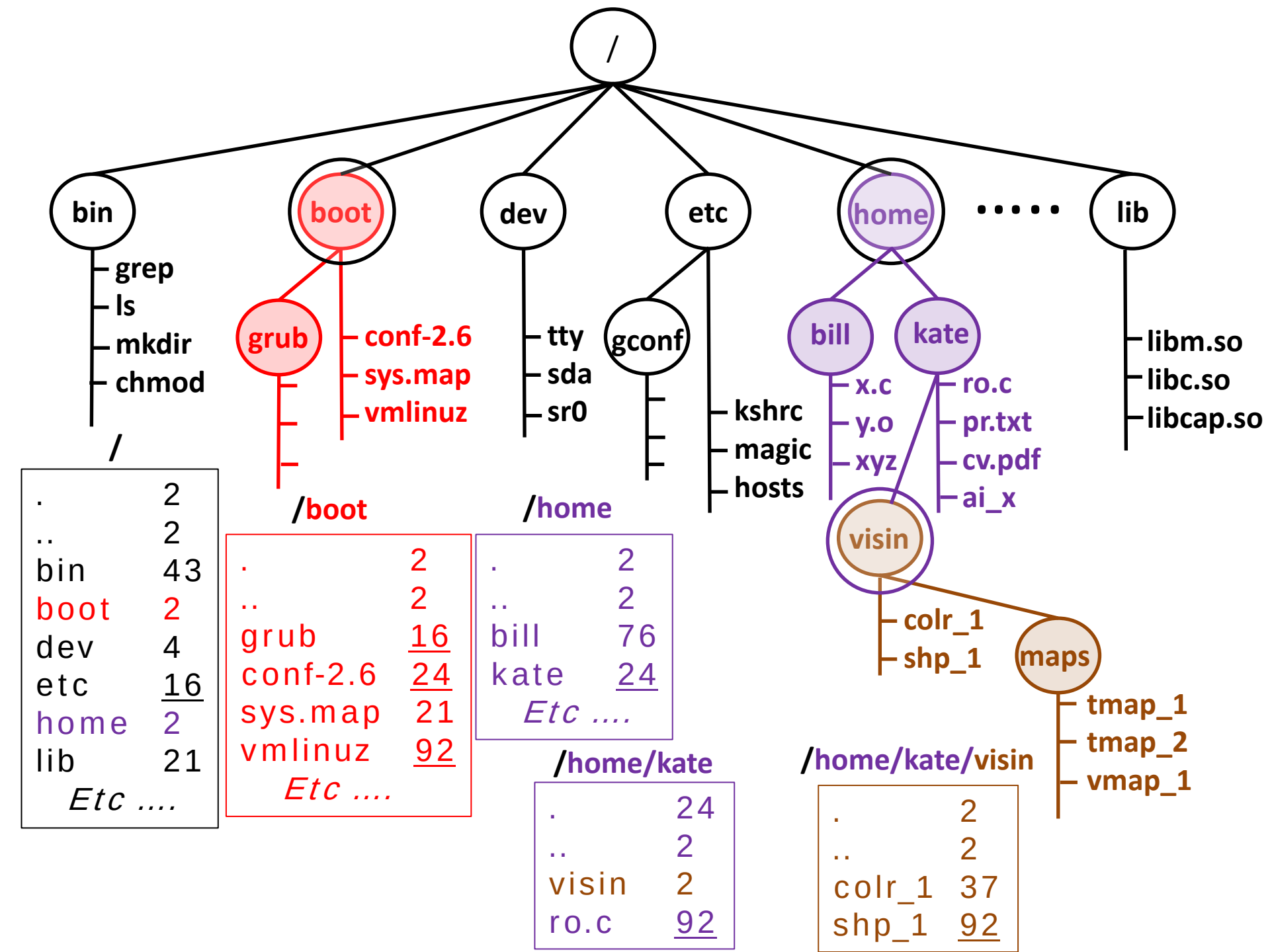
- While recovering after a system failure, the ***e2fsck*** program determines the following cases:
  - ***The system failure occurred before a commit to the journal***
    - Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete; in both cases, ***e2fsck*** ignores them
  - ***The system failure occurred after a commit to the journal***
    - The copies of the blocks are valid, and ***e2fsck*** writes them into the filesystem
  - In the first case, the high-level change to the filesystem is lost, but the filesystem state is still consistent
  - In the second case, *e2fsck* applies the whole high-level change, thus fixing every inconsistency due to unfinished I/O data transfers into the filesystem

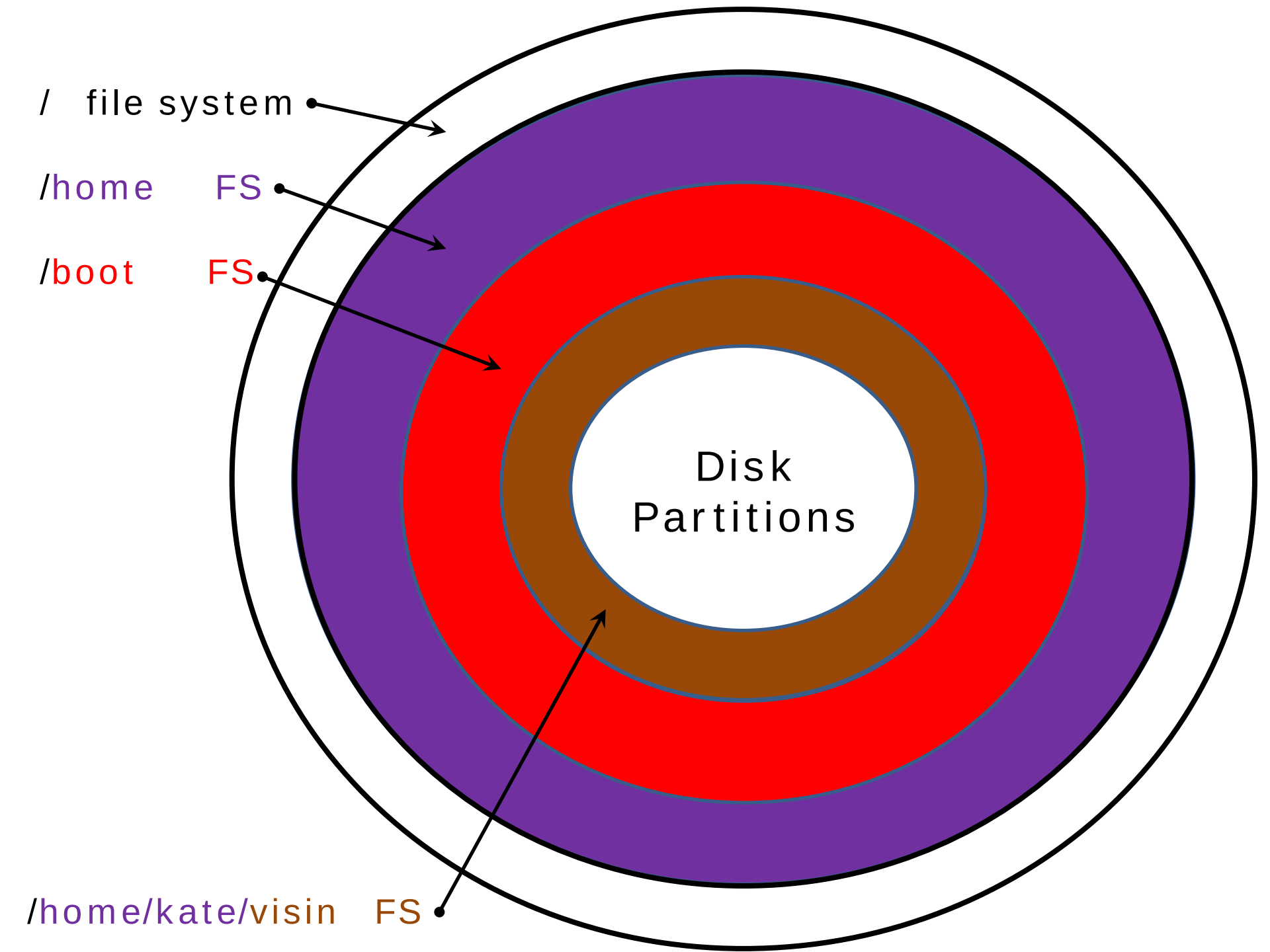
Journaling Options

- The Ext3 filesystem can be configured to log the operations affecting both **metadata** and **data blocks**
- The system administrator decides what must be logged:
  - ***Journal***
    - All data and metadata changes are logged into the journal
  - ***Ordered***
    - Only changes to filesystem metadata are logged into the journal
    - The Ext3 filesystem groups metadata and related data blocks so that data blocks are written to disk ***before*** the metadata
    - This way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file is guaranteed to be fully protected by the journal
    - This is the default Ext3 journaling mode
  - ***Writeback***
    - Only changes to filesystem metadata are logged
    - This is the method of other journaling filesystems and is the fastest

File System Hierarchies

- A typical Linux system generally has at least 2 file systems in its File System Hierarchy
  - A root file system anchored on the name /
    - This file system is mounted at system start-up time
  - A second file system that includes the necessary system boot code
    - mounted on the directory named /boot
- Other local file systems may be mounted into the hierarchy during system initialization
  - These often include file systems mounted on:
    - /tmp
    - /home
    - /var
    - /usr

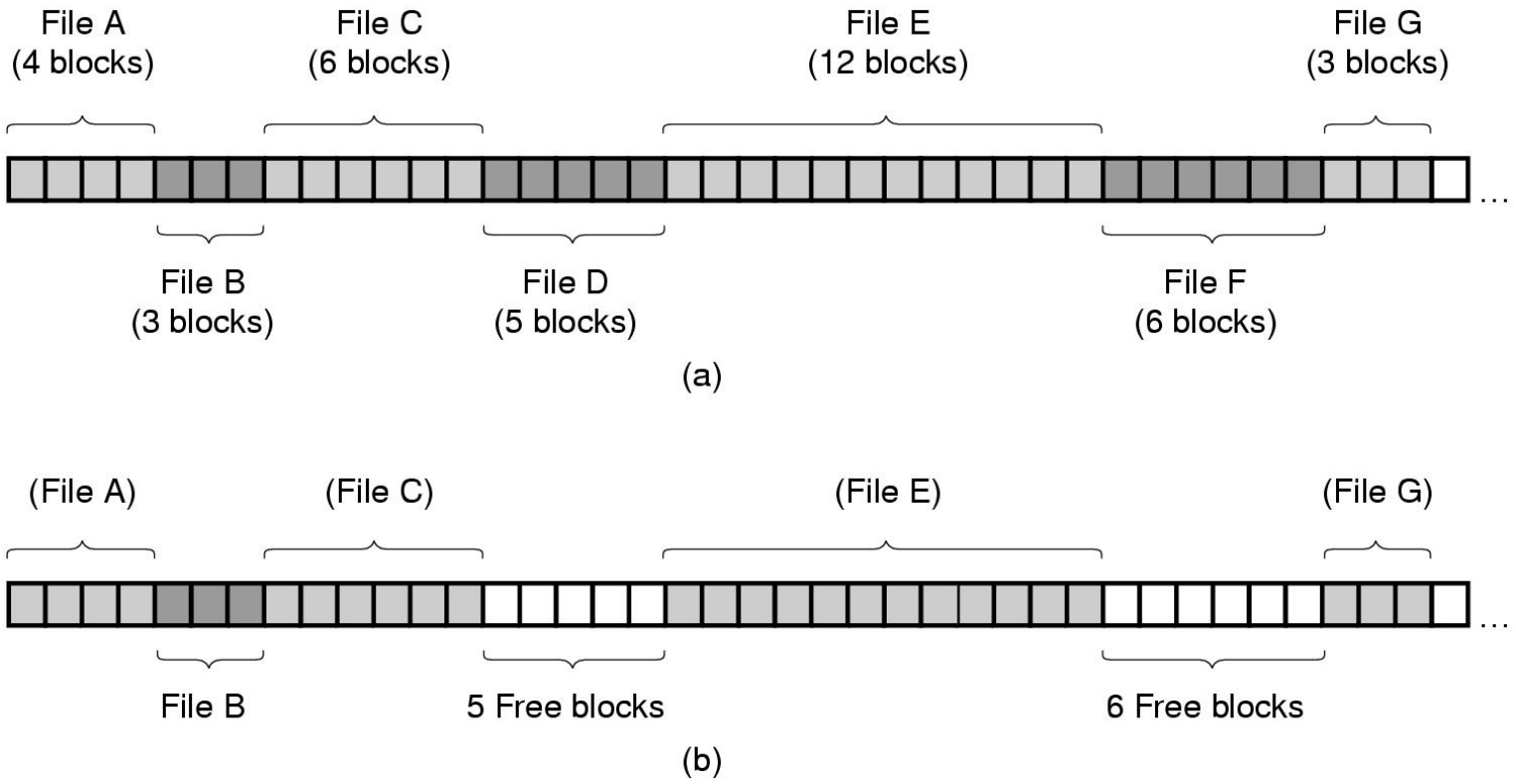

/
bin
boot
dev
etc
home
.....
lib
grep
ls
mkdir
chmod
grub
conf-2.6
sys.map
vmlinuz
tty
sda
sr0
gconf
kshrc
magic
hosts
bill
kate
x.c
y.o
xyz
ro.c
pr.txt
cv.pdf
ai_x
libm.so
libc.so
libcap.so
visin
colr_1
shp_1
maps
tmap_1
tmap_2
vmap_1
/
. 2
.. 2
bin 43
boot 2
dev 4
etc 16
home 2
lib 21
Etc ....
/boot
. 2
.. 2
grub 16
conf-2.6 24
sys.map 21
vmlinuz 92
Etc ....
/home
. 2
.. 2
bill 76
kate 24
Etc ....
/home/kate
. 24
.. 2
visin 2
ro.c 92
/home/kate/visin
. 2
.. 2
colr_1 37
shp_1 92


/ file system
/home FS
/boot FS
Disk
Partitions
/home/kate/visin FS

File System Organizations

- The Ext family of Linux based file systems is an example of an INDEXED organization
- Other organizations include:
  - Contiguous allocation
    - Must keep track of starting block, and number of blocks
    - Provides direct access
  - Linked-list allocation
    - Each allocation unit of an object has a pointer to the next unit
      - Sequential access only
    - File Allocation Table (FAT) linked list organization
      - Makes sequential access a tractable strategy
  - Log based allocation
    - No “in-place” updates, all writes to new storage units

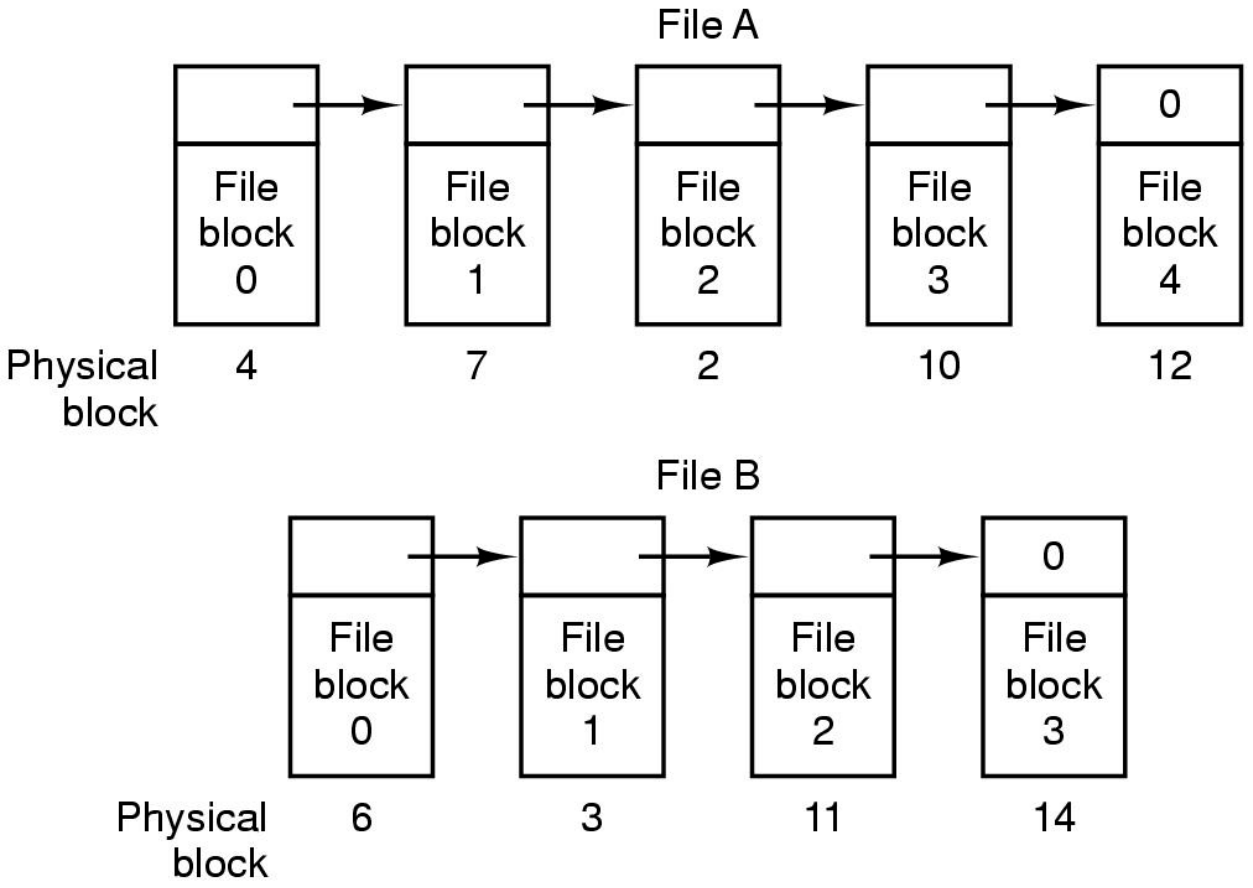
Contiguous Allocation



(a) Contiguous allocation of disk space for 7 files.

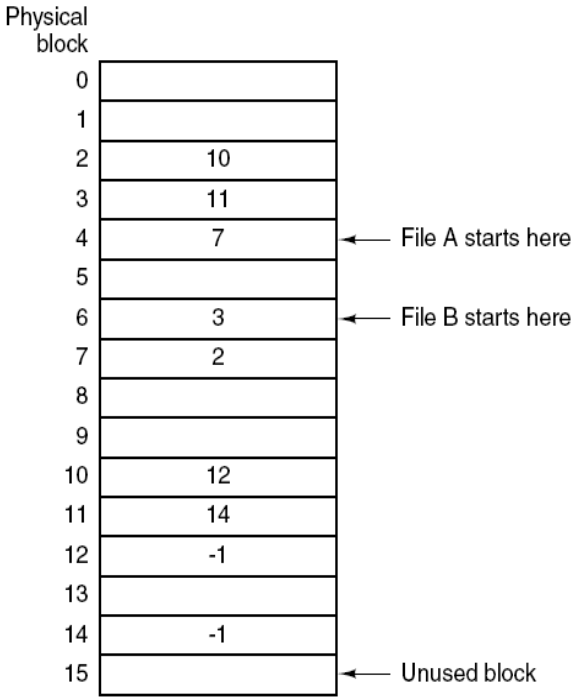
(b) The state of the disk after files D and F have been removed.

Linked List Allocation



Storing a file as a linked list of disk blocks.

Linked List Allocation Using a Table in Memory



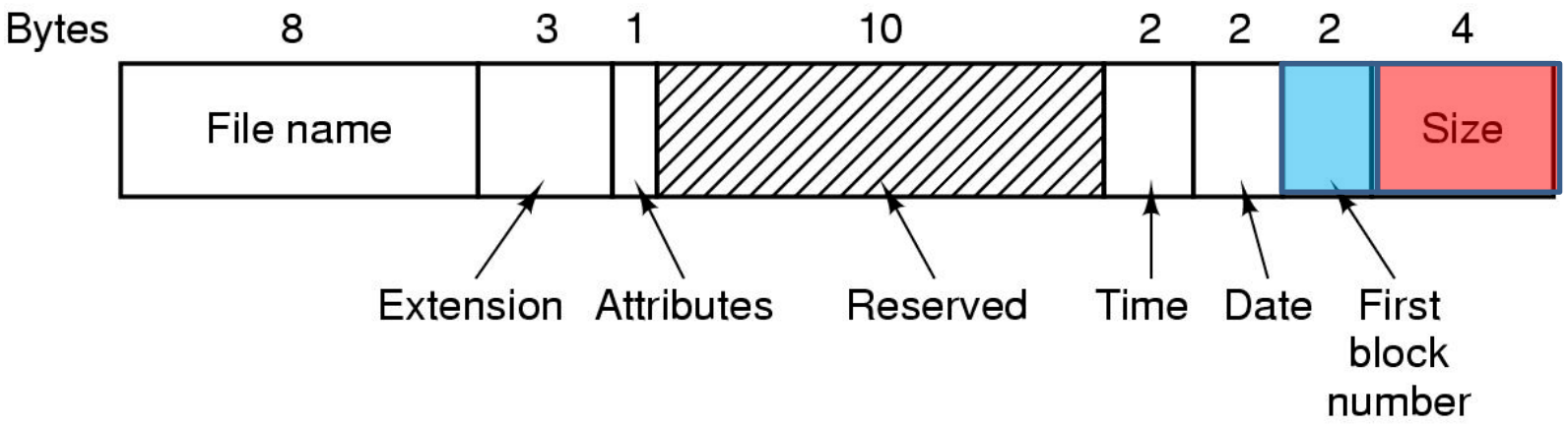
Linked list allocation using a file allocation table in main memory

- Windows **FAT16** file system uses this mechanism
- Table has 2^{16} slots, so can point to 64 K allocation units
- The **size** of an allocation unit (MS calls them clusters) determines the max file size and the max file system size

Microsoft FAT16 Organization

- Since the FAT has only **64 K** entries, entries can be small (an unsigned short)
  - Table can be kept entirely in memory
    - 64 K * 2 bytes/pointer is only 128 KB
  - If allocation unit is a single sector (512 bytes) then max file system size will be
    - **$2^9 * 2^{16} = 2^{25} = 32$ MB**
  - MS allows maximum allocation unit (cluster) size to be 32 KB, so absolute max file size is
    - **$2^{15} + 2^{16} = 2^{31} = 2$ GB**
  - Notice that no more than **64 K objects** can exist in the file system

The MS-DOS FAT16 File System



The FAT16 directory entry

- The **first block number** indexes the table
- The **size** field specifies "last block fragmentation"
  - On average, last allocation unit (cluster) is half empty
  - When clusters are at max (32 KB), this means that we waste about 16 KB per file object
    - With **2^{16}** files objects * **2^{14}** waste = **2^{30}** frag **(half FS size)**

Log Based File Systems

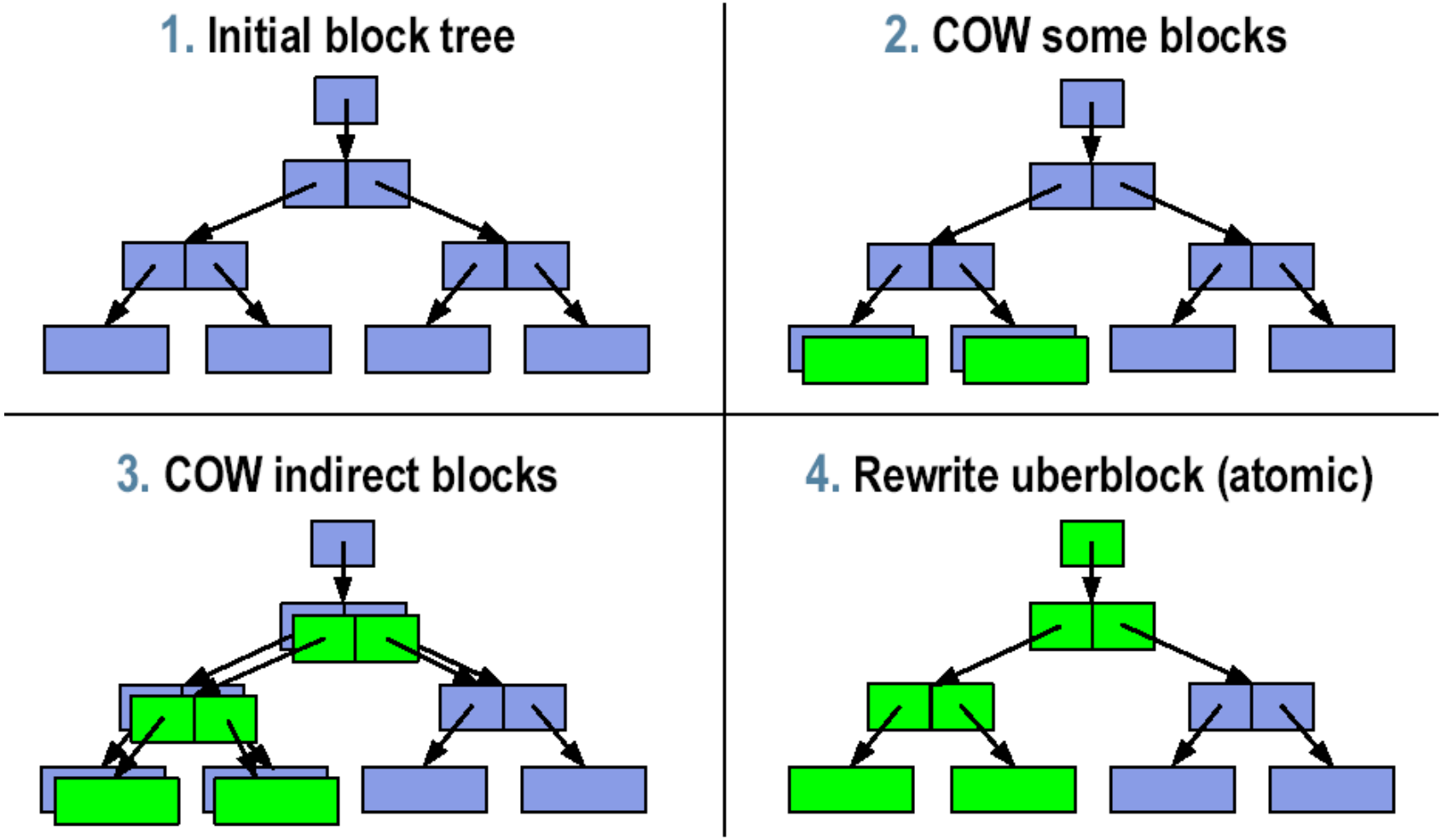
- Log file systems are beginning to displace older file system organizations
- These systems use a **Copy on Write (COW)**, so information is never updated "in place"
  - Guarantees **consistency** under all conditions
- The underlying storage allocation units are written in a circular log like fashion
  - Continuous garbage collection recovers old blocks
  - When anything needs to change, new blocks are located and the entire update is newly written
    - The old, now unused, blocks are scavenged by the GC

ZFS from Sun/Oracle

- ZFS is a storage technology originally developed by Sun Microsystems (now Oracle)
- ZFS Design Principles
  - Pooled storage
    - Completely eliminates the antique notion of volumes
    - Does for storage what VM did for memory
  - End-to-end data integrity
    - Historically considered “too expensive”
    - Current implementation shows that is can be manageable
    - For “simple storage” the alternative is unacceptable
  - Transactional operation
    - **Keeps things always consistent on disk**
    - Removes almost all constraints on I/O order

ZFS (cont'd)

Copy-On-Write Transactions



ZFS (cont'd)

Bonus: Constant-Time Snapshots

- **At end of TX group, don't free COWed blocks**
  - Actually cheaper to take a snapshot than not!